UNIVERSITY OF OSLO
Department of Informatics

Packet Tracing in Simulation Environments

Master thesis

Vladimir Zorin

July 29, 2011

Preface / Abstract

This master thesis is a part of my Master's degree at the University of Oslo. The thesis has been written at Simula Research Laboratory (SRL), where my supervisors Sven-Arne Reinemo and Tor Skeie work. Another advantage of doing my work at SRL is the availability of a small computer cluster on which I could run the tests needed for my thesis.

The goal of my master thesis is providing the Infiniband simulation in Omnet++ with means to simulate real-world network traffic. The master thesis consists of two main parts. The first part is about integrating two simulators – the Infiniband simulation in Omnet++ with LogGOPSim. In the design chapter (Chapter 4) I describe the implementation of the integration, its functionality, the problems experienced and solved during implementation and the integration's effectiveness. In the second part of the thesis, the evaluation chapter (Chapter 5), I describe the process of testing how the integration works and the results of calibrating the input for the simulation. During this explanation I use several simple examples which are supposed to provide clarity in what really goes on during the simulation.

The explanations of the central technologies and terms of this thesis are presented in the two background chapters – Chapter 2 gives a relatively shallow description of the less central terms, while Chapter 3 gives an in depth description of what is most important.

Acknowledgements

There are several people I'd like to thank. First of all I'd like to thank my supervisors Sven-Arne Reinemo and Tor Skeie for their help and support when it comes to both practical tasks and writing process. I would also like to thank Torsten Hoefler, one of the authors of LogGOPSim, for following my progress, interesting discussions and suggestions, and for writing the LogGOPSim tool chain and making it open source. There are three PHD students at SRL who definitely helped me a lot - Wei Lin Guay and Bartosz Bogdanski were very patient helping me with Omnet++ and cluster related issues, while Ernst Gunnar Gran gave me one very good idea on optimizing the integration. Special thanks go to Simula Research Laboratory for providing me with the necessary means to write this thesis.

Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction

The current trend is that the growing demand for computing capacity is met through *parallelism*. One form of parallelism is the machine level parallelism, i.e. dividing the processing between multiple machines. Thus the role of interconnection networks is growing.

Simulations are useful tools in the design of High-Performance Computing (HPC) systems which may consist of thousands of processing nodes interconnected by a network. The purpose of simulations is optimizing the network topologies, switch and network adapter architectures and parameters, scheduling and routing policies as well as flow- and congestion control mechanisms. [41]

Some simulators can perform detailed cycle-accurate simulations at instruction level and are excellent for evaluating processor performance. However, the high level of detail prevents scaling of this type of simulation to large number of nodes.

Simulators with a higher level of abstraction are more scalable and make it possible to model the relatively large (thousands of nodes) interconnection networks of HPC systems. The drawback here is that such simulators typically simulate some synthetically generated network traffic [42 p. 479] which can be sufficient for determining such characteristics of the loaded network as throughput or latency, but can not guarantee high performance of a specific application in the real world. This synthetic network traffic issue is present in the Infiniband Model in Omnet++ (later referred to as *IB Model*). Solving this drawback is the main problem setting of this master thesis. The IB Model is a relatively high abstraction level simulator of the link layer of an Infiniband subnet.

It was chosen to use the so called *trace-driven simulation* approach. With this approach the behavior of the simulated network nodes is determined by a predefined *schedule.* The schedule typically consists of records representing the node's computation and sending/receiving of messages. There is also a mechanism of ensuring the desired sequential order of the records (the *dependency* mechanism).

The Message Passing Interface (MPI) (Section 2.7.5) is standard for many applications running on the HPC systems. In this master thesis we will be using simulation schedules based on the traces of MPI calls – each call to an MPI function is logged, and the simulation replays this log.

1.1 Methods Used in This Thesis

As already mentioned the main goal of this master thesis is providing the IB Model with the means of simulating the real world network traffic instead of synthetic one. The way of doing this is integrating the IB Model with another simulator called LogGOPSim. The LogGOPSim tool chain (Section 3.1) provides means of producing and parsing MPI traces and converting them into the simulation schedules. These schedules are then simulated by LogGOPSim with the IB Model as the link layer.

When a relatively long simulation is giving unexpected results, it may be extremely hard to find the cause of the problem due to the simulation length. Series of short simulations are used for more extensive testing of the integration and achieving complete understanding of the problems present and the ways of solving these problems. The correctness is evaluated by comparing the

simulation log to the MPI trace on which the simulation was based for short tests. For long tests we use the comparison of real application running time to the simulated running time. The efficiency of the integration is measured using a set of practical tests.

1.2 Short User's Guide

This thesis consists of six chapters including this one. The second chapter contains relatively high level background information about the related technologies. Chapter 3 is the second background chapter and gives a more in depth presentation of the two network simulators which are central in this thesis. There are two main chapters. Chapter 4 describes the design of the integration of the two simulators presented in chapter 3, as well as the evolution and efficiency testing of this design. Chapter 5 presents several simulations run using the integration of the two simulators, describes the encountered problems and proposes solutions to some of the problems. The final sixth chapter is the summary of the thesis; possible future work is also proposed there.

The source code for the thesis can be found at http://heim.ifi.uio.no/vladimz/code/ .

Chapter 2: Background

In this chapter several technologies and topics which are important for this master thesis are presented. This presentation should give the reader a **general** understanding of the topics.

It may be useful to understand what a model is, because the LogP model family will be used and mentioned a lot in this thesis. Simulation environments are also quite central, so an explanation of network simulations will be given with examples. The Omnet++ simulator will be used for simulating the Infiniband networks during the work on this thesis, so both these technologies are presented too. Then there is a brief summary of parallel computing with slightly more focus on the Message Passing Interface. And finally I explain what tracing and profiling are.

2.1 Models

A *model* is anything used to represent something in the real world. Studying a model can help understand the real world. Modeling can be used for planning or analysis of whatever the model represents. An *analytical model* is an equation (or a set of equations), involving the (important) variables describing whatever is modeled, possibly omitting the less important variables. Let's look at a couple of simple examples of analytical models:

Example 1: If we have $100 on a bank account with an interest rate (I) of 2%, an analytical model of how the amount (A) of money changes on the account after n years would be $A = A_{initial} * (1+I)^n$. In other words after 5 years we would have $100 * 1.02^5 = \$110.4$ on the account.

Example 2: We are transferring data on a 1Mbit/s channel. How much time will it take to transfer a 1MiB file? If we use the simplest model possible – transfer time = amount of data divided by bandwidth, we'd find out that transferring 8,388,608 bits at the speed of 1 million bits per second would take about 8,389 seconds. However, we could add numerous other variables to our equation to make this simple model more realistic. For instance we could take into account that most probably our data will be divided into packets, and packets will have headers, so the actual amount of data that needs to be transferred is higher, and therefore the transfer will take more time.

The analytical model in the second example above would have to be more complicated and include more variables to give a realistic representation of the real world. If the real-world phenomena is too complex, the model approach may be unsuitable for studying this phenomena. A model of a complex system may be impossible to solve mathematically. [1]

2.2 The LogP Model Family

A communicating system may be a parallel application (Section 2.7) where the different processes (parts of this application) communicate with each other using for example MPI (Section 2.7.5).

An example of such a parallel application can be Omnet++ presented in Section 2.5, which is capable of running parallel distributed simulations. During a parallel simulation in Omnet++ the functionality of different Omnet++ modules resides in the different processes that the parallel simulation consists of. There are several conditions that must be met for this: no global variables, no member access between modules mapped to different processes, all communication between

modules should happen through messages, etc. [5 section 14.3]

The communication in a parallel system can be characterized by a set of parameters. Such a set of parameters is called a *model*.

The original LogP model, as it's name hints, describes the communicating systems using four parameters: L, o, g and P where

- 'L' stands for maximum latency between any two processors in the system
- 'o' stands for cpu overhead per message
- 'g' stands for time (gap) between two message insertions into the network
- 'P' is the number of processors in the system.

According to this model up to L/g packets can be in flight between the two end nodes. Contention is not taken into account in this model. The LogGP model adds an additional parameter G:

- 'G' is the gap per byte of a long message.

Since most networks are able to transmit large messages relatively fast due to fragmentation and reassembly in hardware, the cost per byte metric 'G' is more accurate than using just 'g' and modelling multiple small messages in LogP. So the LogGP model uses two bandwidths: L/g for small and L/G for large messages.[7]

However, though the LogGP model reflects the advantage of special support for large messages, it doesn't reflect the need for synchronization between the sender and the receiver of a large message. In many MPI implementations different protocols are used for sending messages of different lengths. Therefore, in the LogGPS model another parameter, S, is introduced:

- 'S' determines the message-size threshold for synchronizing sends.

When a message is larger than S bytes the so called *rendezvous* protocol is used, where the sender checks with the receiver whether sufficient buffer space is available before sending a large message. This is done using small control messages. [7] [8]

One shortcoming of the LogGPS model is that it models only a constant processing overhead per message send, independent of message size. This shortcoming is eliminated in LogGOPS model by a new parameter O:

- 'O' is cpu overhead per byte

The LogGOPS model is used in the LogGOPSim simulator by Torsten Hoefler and Timo Schneider. This simulator is presented in detail in Chapter 4.

2.3 Simulations

Simulation is the imitation of the operation of a real-world process or system over time. Simulations are suitable for study and experimentation with complex systems, verifying analytic solutions. Simulations are relatively hard to construct, as the constructor needs to understand the work-flow of whatever is simulated. Simulations should not be used when the phenomena studied can be modeled analytically, or where common sense can be used. [1: sections 1.0-1.2]

Simulations can be divided into *continuous* and *discrete-event simulations*.

- "*A continuous simulation concerns the modeling over time of a system by a representation in which state variables change continuously with respect to time.*" Usually differential equations are used in such simulations. [39] This type of simulations can be used for example in computer games. [38]
- A discrete-event simulation is modeling of systems in which the state variable changes only at discrete points of time. [1: section 1.10]

In this master thesis discrete-event simulations are most central. A discrete-event simulation involves an *event list* or *event queue* ordered by time (a priority queue can be used for implementation). An *event* is an occurrence that changes system state. A *system* is a collection of entities that interact together, while a *model* is an abstract representation of this system. The s*ystem state* is the collection of variables needed to describe the system at any time. There is also a variable representing the current *simulated time*. The simulated time is advanced to an event's time when the event is popped from the event queue. [1: section 3.1][2][3]

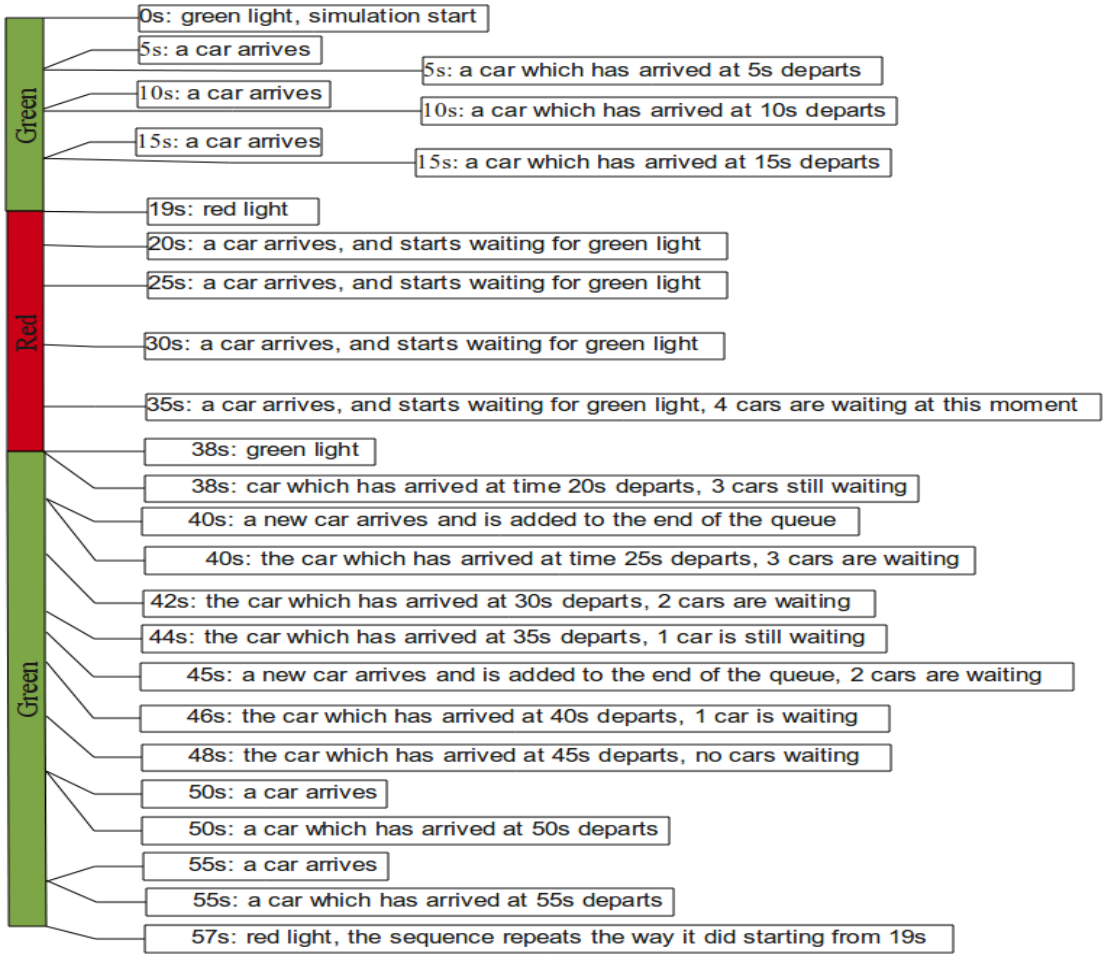

Figure 2.3.1 event sequence in the discrete-event simulation example

Example: Let us look at an example which at first may seem very simple and suitable for modeling with an analytical model, but when we try to make a simulation for it, it turns out to be quite complex. There is a traffic light with two states: red and green. The state transition happens every 19 seconds. At simulation start the traffic light is green. The cars are arriving with a rate of one car every five seconds. When the traffic light is green one car can drive through every two

seconds. The events in our simulation would be car arrival/departure from the traffic light and traffic light state transitions. The adjustable parameters would be arrival/departure rates and red/green light durations.

The event sequence in the discrete-event simulation of the traffic light could be something like the one presented on Figure 2.3.1. How events are added to the event queue depends on the implementation of the event handlers. For example in the car arrival event handler we could do the following.

- Check that the light is green, and no cars are waiting and the last car has departed at least 2 seconds ago, then the new departure event can be added to the event queue with time stamp equal to current simulated time. (immediate departure)
- If the light is green, and no other cars are waiting, but the last car has departed less than 2 seconds ago, the new departure event should be scheduled so that it occurs 2 seconds after the previous departure. Notice, that the situation when the departure event occurs after the light shifts to red has to be handled in the departure handler.
- If the light is green, but there are other cars currently waiting in front of the just arrived car, the waiting counter should be incremented.
- We should also schedule a new arrival event with time stamp 5 seconds in the future.

The pseudo-code for all four event handlers can be found in **Appendix A**. The simple traffic light situation experienced by all of us every day resulted in a relatively complex simulation. When looking at the presented example it is easy to imagine which other features could be added to it making the simulation more realistic and more complex. Modeling for example a computer network in detail may be a much more challenging task.

2.4 What is a Network Simulation?

According to [4] a network simulation is a "*technique where a program models the behavior of a network either by calculating the interaction between the different network entities using mathematical formulas, or actually capturing and playing back observations from a production network*." So if we have a real network consisting of some devices like hosts, links, switches and routers a simulation would be a program/logic attempting to recreate the interaction between these devices. Most network simulators, including Omnet++, use discrete event simulation, where there is a list of pending events typically sorted by time at which a certain event is supposed to happen.

Let us look at a simple example. We want to simulate a network of two hosts connected with a single link and playing ping-pong with data packets. In our example it takes 1 ms to send a packet between host A and host B. When one host receives a packet from the other it should immediately send it back. So if we want to simulate this (and assuming our simulator is object oriented) we would need two objects to represent our two hosts, in addition to some control logic. The control logic would be simple: when a host sends a packet we add an event to the list and mark it with the time 1 ms away from the current time (the packet arrival time). Then retrieve the event with the earliest time and call the "receive" function belonging to the receiving host object. In this function we simply call "send" and the whole thing is repeated again, except that the sender and receiver switch places. This event sequence is illustrated in Figure 2.4.1.

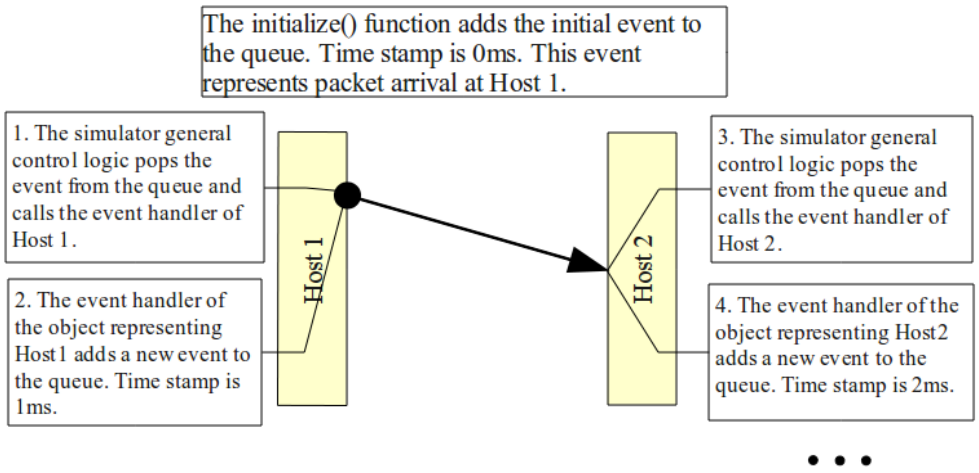

Figure 2.4.1 A simple network simulation example

It is obvious that the simple example above can be done more complicated. For example we could add more hosts (and links connecting them). We could also add some sort of addressing/identifying these hosts. The packets can actually contain the destination address (or, to be more precise, the event representing a packet on the way would contain information about which object's receive function should be called when time for this event comes.) In addition to the hosts another type of network nodes – switches or routers could be added so that we could form topologies. The list of features that could be added is long. For a simulation to reflect the real world network in a realistic manner, a lot of features of the real network have to be taken into account.
[4]

2.5 Omnet++

Omnet++ is an object oriented modular discrete-event network simulation framework. The IB Model (Section 3.2) which is quite central in this master thesis is written using this framework.

The Omnet++ framework provides API and tools for writing simulations. The internal logic provided includes adding/retrieving events from the event list, basic internal functions for the network nodes (such as getId()), etc. Omnet++ also provides several simulation model components, such as generic network nodes, links and messages. A simulation model is composed of such components. The network nodes are called modules.

Modules can be connected to each other via ports or combined to form compound modules (for example a switch compound module can consist of simple modules “input buffer”, “output buffer” and a “packet arbitrator” in-between). See Figure 2.5.1 for a schematic overview. All modules that are not compound are called simple modules. The depth of module nesting in compound modules is unlimited. Modules communicate using messages, which can carry arbitrary data structures. Messages can travel both through links, or directly. Simple modules of the same type may be initialized with different parameters to customize module behavior. The logic of simple modules is programmed in C++ by users.

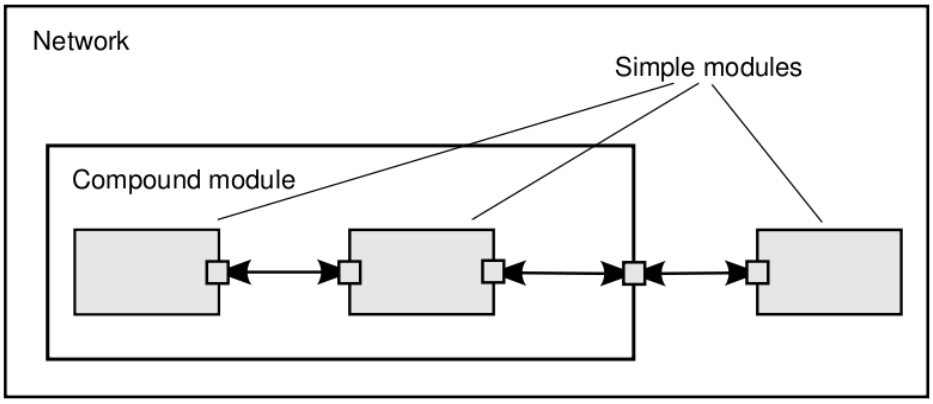

Figure 2.5.1 A schematic view of a network consisting of one compound and one simple module.

The user has to define the topology of the network, the network nodes (for example number of ports) and the detailed functionality for the nodes. By node functionality I mean for example what a node does when it receives a packet. Nodes can use self messages to control the internal functionality/timers in addition to functionality triggered by external messages ("packets") received from other nodes (although both self messages and packets are just events in the list). This gives the user the flexibility to simulate basically any network. [5]

All simple modules must have at least two functions: Initialize and HandleMessage. In the Initialize function it is usual to initialize the simple module's data-structures and schedule the initial event(s) (messages) to start the activity. It is the internal logic of Omnet++ that pops these events from the event list at the appropriate simulated time and calls the HandleMessage function of the appropriate simple module.

There is a set of examples and tutorials coming with Omnet++ distribution. One of the example sets coming with version 4.1 is called "tictoc". The first example in this set is exactly the same as the simple network of two nodes from the example in the Network Simulations section (Section 2.4). There is one simple module (described by a class in C++) representing a network node. During simulation there are two instances of this simple module. Each of the nodes has one input and one output gate; the first node's output is connected to the second node's input, and vice versa. The two unidirectional links have the latency property of 100ms (it was 1ms in my example from the Network Simulations chapter). In the Initialize function belonging to the first node the initial message is sent. The only thing done in the HandleMessage function which is called on message reception is sending the received message out through the output gate.

2.6 The Infiniband Architecture

The Infiniband Architecture (IBA) is an industry-standard architecture for server I/O and inter-server communication [40].

2.6.1 Infiniband Concepts

An Infiniband subnet consists of switches and end-nodes connected with links (copper or fiber). Several IB subnets may be connected by IB routers into a larger network. Within a subnet one end-node or switch acts as a centralized subnet manager. The links are interfaced by the end-nodes with network cards called Channel Adapters (CA). Each end-node port or a switch has a 16-bit address called Local Identifier (LID); routing between subnets is based on a 128-bit

Global ID (GID). [6]

2.6.1.1 Switches

The basic task of switches is forwarding packets: receiving a packet with a given destination, performing a forwarding table lookup and sending the packet to the output port based on the lookup result. The number of ports in a switch can not be larger than 256 [36], typically much smaller – the largest IB switches in current equipment have 36 ports. [6]

2.6.1.2 End-nodes

End-nodes are hosts or devices like storage subsystems, etc. They act as communicating parts in a network. End-nodes generate and consume traffic. [6]

2.6.1.3 Routers

Routers forward packets from one subnet to another. While forwarding in switches is based on LIDs, forwarding in routers is based on the global 128-bit addresses. [6]

2.6.1.4 Links

Links interconnect channel adapters, switches and routers. A link can be copper or optical. The status of the link can be determined via the device on each end of the link. The following link widths and speeds are specified:

	SDR	DDR	QDR
1x	2.5Gbit/s	5Gbit/s	10Gbit/s
4x	10Gbit/s	20Gbit/s	40Gbit/s
8x	20Gbit/s	40Gbit/s	80Gbit/s
12x	30Gbit/s	60Gbit/s	120Gbit/s

Table 2.6.1.4 IB Link properties [43]

The actual devices don't necessarily support all the combinations, and the actual bit rates are 80% of the line rates. [40]

2.6.1.5 Channel Adapters

There are two types of CAs: Host Channel Adapters (HCA) and Target Channel Adapters (TCA). The former, as the name suggests is used for hosts. The later is used for peripheral devices. The difference is that HCAs have a collection of features available for applications running on hosts through functions, while the TCAs don't have a defined software interface. Currently the CAs are cards attached to a standard I/O bus. [6]

2.6.1.6 Subnet Management

Every IBA subnet must contain at least one subnet manager (SM) residing on an end-node or a switch. An SM starts in *discovery phase* when it discovers all the switches and hosts in the subnet. If other SMs are discovered, a negotiation of who should be the master SM takes place. When this is done, master SM enters the *master phase* during which it assigns LIDs, configures switches and ports and calculates forwarding tables. The last phase is called *subnet phase*, when the subnet is ready for use. During the subnet phase an SM periodically checks the subnet for

topology changes, and reconfigures the subnet if necessary. [35]

2.6.1.7 Queue Pairs

The QPs are a **Transport Layer** concept. A QP is a virtual interface provided by hardware to the consumer. A consumer may be any application operating above the Transport Layer of the OSI model, for example an MPI application (Section 2.7.5). A queue pair consists of a Send and a Receive Queue. The send and receive work requests are posted by the consumer into the respective queue. The QPs are not created, operated and destroyed by the consumers directly, but by using the provided functions.

The service provided by a QP may be connection oriented, when two QPs are tightly bound to each other, or connectionless (datagram oriented). The service can also be reliable (acknowledged) or unreliable (unacknowledged). Raw datagram type of service means that data can be sent to non Infiniband destinations (naturally it is not reliable or connection oriented). [6]

2.6.1.8 Virtual Lanes

A Virtual Lane is a **Link Layer** concept. IBA switches support between 2 and 16 Virtual Lanes (VLs). Virtual Lanes provide support for independent data streams on the same physical link. They are used for deadlock avoidance and prioritization/segregation of traffic classes. The two required VLs: VL0 and VL15 are for normal data traffic and for subnet management traffic respectively. Presence of more than one data VL is optional. [6]

There are separate buffering resources and flow control for each data VL. That is when a data packet arrives at a port it shall be placed in the buffer associated with that input port and VL field in the packet. This means that excessive traffic on one VL does not block traffic on another VL. Packets on VL 15 are not subject to flow control, and always have the highest priority. [6 sections 7.6.3-4]

2.6.1.9 Service Level

Service Levels (SL) are used to identify different flows within an Infiniband subnet. Unlike the VLs, the SL is never changed while a packet travels through a subnet. SLs are intended to aid in implementing Quality of Service related services. The SL to VL mapping mechanism is used for changing packet VL while it crosses the subnet. This is needed if the next link in the packet's path doesn't support a certain VL or if two input streams are destined for the same output link and also use the same VL (so that the two streams stay separated). [6]

2.6.1.10 Flow control

Credit based flow control is utilized in IBA at the link layer – in other words the flow control is not end-to-end. Flow control is VL based (only data VLs). A sender does not send anything unless it has credits provided by the receiver (the node on the other side of the link). This way the packets are never dropped because of overflow. Each port must advertise the number of credits (input buffer space) available for each data VL using flow control packets.
As already mentioned packets never get dropped in Infiniband (unless a bit error occurs and CRC check fails.) This is an advantage over for example conventional Ethernet, where packets may often be dropped due to queue overflow. [6][10]

2.6.1.11 Congestion Control

Congestion arises when an application sends more data than switches or routers can accommodate. Generally congestion may lead to packets being delayed or dropped (the later doesn't happen in Infiniband due to lossless flow control).

Congestion control is an optional feature in the Infiniband Architecture. If it is implemented, the switches have the responsibility of discovering congestion. When the amount of packets in an input buffer reaches some value (for example when a buffer gets 60% full) the switch enters congestion state. When congestion is detected by a switch, there is a chance that the packets causing congestion get marked. The marking happens by Forward Explicit Congestion Notification (FECN) bit being set in the packet header. When the destination node discovers that this bit is set, it sends a congestion notification packet (or in case of reliable connection an acknowledgement) back to the source with Backwards Explicit Congestion Notification (BECN) bit set which causes the source to temporarily reduce packet insertion rate. Several parameters determine when switches detect congestion, at what rate the switches will notify destination nodes setting FECN bit, and how much and for how long a source node contributing to congestion will reduce its injection rate. [11][6]

2.6.2 Infiniband Layered Architecture

Infiniband provides a range of services up to the Transport Layer of the OSI model as shown in Figure 2.6.2.0. IBA operation can be described as a stack of layers, where each layer depends on the service provided by the layer below, and provides service to the layer above.

*2.6.2.1 The P*hysical *Layer*

The *physical layer* is the lowest layer. It specifies how bits are placed on the wire, how symbols are formed (symbols like start/end of packet, data symbols, space between packets), synchronization method, etc. All this is specified in detail in volume 2 the of Infiniband Architecture Specification – however going deeply into this is not needed in this master thesis.

*2.6.2.2 The L*ink *Layer*

The *link layer* describes the packet format, flow control (Section 2.6.1.10) and how packets are routed within a subnet. There are two types of packets: Link Management Packets and Data Packets. The link management packets are used for maintenance of link operation – sending flow control credits, maintain link integrity, negotiate operational parameters between ports (parameters like bit rate, link width, etc). These packets aren't forwarded to other links. Data packets, as their name suggests carry data. They also have several headers, some of which might or might not be present. The link layer header is called the Local Route Header. It is always present, and contains the local source and local destination ports, Service Level (Section 2.6.1.9) and Virtual Lane (Section 2.6.1.8). Source and destination fields contain 16-bit Local IDs (LIDs) assigned to each port by the subnet manager. The VL field may change while the packet travels through the subnet, while the other fields stay unchanged. There are two CRC fields: one covering all unchanged fields, and the other covering all fields of the packet, which make it possible to check data integrity both end to end and hop by hop.

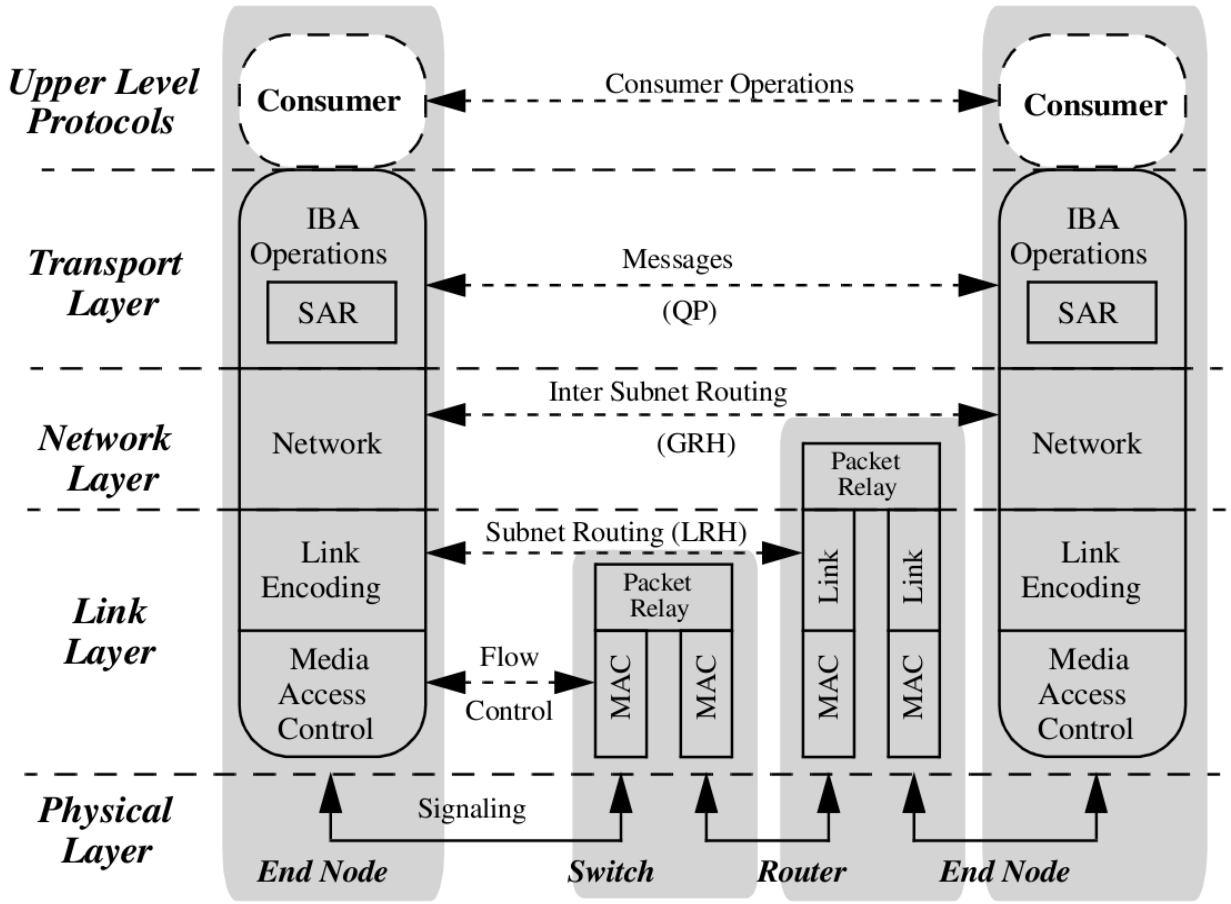

Figure 2.6.2.0 IBA Layers [6]

*2.6.2.3 The N*etwork *Layer*

The *network layer* describes the protocol for routing a packet between subnets. A packet traveling between subnets has a Global Route Header (GRH) containing the 128-bit Global ID (GID). GID is in the format of an IPv6 address. For such packets the LRH is replaced in each subnet traversed to contain the LID of the edge router. The last router replaces the LRH using the LID of the destination.

*2.6.2.4 The T*ransport *Layer*

The *transport layer* header is called Base Transport Header (BTH). It is present in all packets except raw datagrams. The transport layer protocol is responsible for delivering packets to the proper Queue Pairs (QP) (Section 2.6.1.7) and instructing the QPs on how to process the packet's data. The messages which are larger than the MTU are also segmented into multiple packets in the transport layer (and reassembled back at the destination). The transport layer provides several operations: Send, RDMA Write, RDMA Read and Atomic. There are various Extended Transport Headers optionally present depending on the operation being performed. The transport layer communication in Infiniband may be reliable connection oriented (RC), reliable datagram (RD), unreliable connection oriented (UC), unreliable datagram (UD) and raw datagram. During unreliable service no acknowledgements are sent, there are no packet order guarantees and on error the packets (and hence the whole messages) are silently dropped. During reliable service

acknowledgements are sent for the successfully received messages, and packet order guarantees are given (due to packet sequence numbers). For connection oriented service each QP is associated with one remote consumer, which requires certain information exchange between the communicating parts. This is not the case for the datagram (connectionless) service. Raw datagrams are used for encapsulating either Ethernet or IPV6 packets.

A send operation is for moving a single message to the remote QP. The size of the message is up to 2GiB. Naturally the message may be larger than a single packet (PMTU) for all kinds of reliable and connection oriented communication – in such case the message will be segmented into multiple packets. On the other hand, unreliable datagrams may consist only of a single packet.

RDMA write operation is used for writing into the virtual address space of a destination node. The data is written into the memory allocated by the destination QP. The destination must provide a 32-bit key to the source, which includes this key in the header of the first (or only) packet of the operation (just like Send, this operation may require several packets.) The buffer's virtual address and length must also be provided by the destination. RDMA read is very similar to write.

The atomic operations execute a 64-bit operation at a specified address on a remote node. The mechanism is similar to RDMA operations. We are guaranteed that the given address is not accessed by other QPs between the read and the write. The two atomic operations defined in IBA are Fetch&Add (i.e. increment) and Compare&Swap (used for mutex).

[6]

2.7 Parallel Computing

Parallel computing is a form of computation in which multiple calculations are carried out simultaneously. The premise for this is the possibility to divide a large problem into smaller ones. There are several technologies which make parallel computing easier – some examples are given below. OpenMP, Shared memory and message queues are used for communication between processes or threads running on the same computer, while MPI is mainly for communication between processes running for example on nodes of a computer cluster. [12]

2.7.1 OpenMP

OpenMP (Open Multi-Processing) is an implementation of multi-threading. A master thread starts a certain number of slave threads and a task is divided among them. There are several ways of dividing a task between several threads, for example splitting up loop iterations among the threads, assigning independent code blocks to different threads or serializing a section of code. OpenMP also provides a variety of synchronization constructs, like critical sections, atomic operations, barriers, etc. [13][14]

2.7.2 Shared Memory

Shared memory can be used to implement communication between several processes (memory shared by threads within the same process is usually not called shared). Shared memory is memory that may be simultaneously accessed by multiple programs. No synchronization means

are provided. [15] Shared memory is used in the implementation of the integration of LogGOPSim and IB Model.

2.7.3 Message Queues

Two or more processes can exchange information via access to a common *message queue* (aka *mailbox*). This mechanism is built into Linux. Communicating processes must share a common key to gain access to the queue. The message-passing module belonging to the OS handles access to the queue and provides an interface for sending and receiving the messages, and controlling the queue. [16] Message queues are heavily used in the implementation part of this thesis.

2.7.4 Programming Languages

There is also a number of *programming languages* supporting concurrent programming. The most well known is probably Java. Erlang is an example of a proprietary general-purpose concurrent programming language and runtime system. [17] There are over 50 programming languages listed in the Wikipedia article about the concurrent computing [44].

2.7.5 Message Passing Interface (MPI)

MPI is an API specification that allows processes to communicate with one another by sending and receiving messages. It is typically used for parallel programs running on computer clusters. Both point-to-point and collective communication is supported (a procedure is collective if all processes in a process group need to invoke it). [18] [19] This technology is central in my master thesis, so I'll describe it in a relatively detailed way.

Point-to-point operations are data exchange operations between process pairs (send/receive).

Collective operations involve communication among all processes in a group (either the entire process pool or its program-defined subset). A typical collective function is MPI_Bcast, which broadcasts data from one node to all nodes in the group. An opposite of broadcast would be MPI_Reduce which takes data from all processes, performs some operation on it (like sum or product) and sends the result to a single node. Mpi_Allgather gathers data from all tasks and distributes it to all. MPI_Alltoall is an extension of MPI_Allgather. During MPI_Alltoall each process sends distinct (not the same to all) data to each process. [20] [19]

MPI also provides functions for synchronizing the nodes (for example MPI_Barrier) and obtaining network related information like the number of processes, current process ID, etc.

MPI belongs in layer 5 of the OSI Reference Model. Most MPI implementations consist of a specific set of routines (an API) callable from Fortran or C. I'm using the Open MPI[1] implementation in my theses. The most recent version of the MPI standard (MPI-2.2 aka MPI-2) specifies over 300 functions. [19]

An MPI program consists of autonomous processes, executing their own code, which is not necessarily identical. The processes communicate via calls to MPI communication primitives. Typically each process executes in its own address space, or even on a separate node of a cluster. [18]

1 http://www.open-mpi.org/

2.7.5.1 Eager and Rendezvous Protocols

In MPI two types of protocols are used depending on the size of a message to be sent. The relatively small messages are sent unsolicited, i.e. a message can be sent before receiver calling a receive function. When a message is relatively large, rendezvous protocol is used, when a message is sent only when receiver is ready to accept it.

While an MPI application is running, sends and receives rarely match in time. In real MPI implementations there are two queues: receive queue and unexpected queue (aka early arrival queue). When an MPI receive function is called the unexpected queue is searched first for the message. If the matching message entry is found in the unexpected queue, the entry is removed, and we proceed. If the entry is not found, a new receive entry is posted in the receive queue. When a message actually arrives the receive queue is searched first for the matching receive entry. If the entry is found, it is removed, and we proceed (Msg_arrived()). If the matching entry is not found a new entry is added to the unexpected queue. [21]

When a message size exceeds a certain limit, the so called rendezvous protocol is used. Before sending data, the sender sends an envelope to the receiver. The envelope contains information needed for matching by the receiver and the message ID. The envelope is matched against the receive queue (see paragraph above). If the matching entry is found in the receive queue, a notification is sent to the sender, so that the data can be sent, otherwise an entry is inserted into the unexpected queue and the data is not sent before a receive request is posted. [22]

2.8 Profiling and Tracing

Profiling and tracing are two terms which are used to refer to two different kinds of performance analysis. In profiling we produce some general statistics, like total time spent inside MPI functions (Section 2.7.5) or the total amount of data sent. In tracing the event history is logged, which means that we get lots of details, but also large amounts of data. In both cases, however, the data is produced during program runtime, as opposed to static code analysis. [23][24]

The main way of using profiling or tracing is to intercept function calls from user code. The MPI-2.0 specification defines a mechanism through which all of the MPI defined functions may be accessed with a name shift. This means that all the MPI functions, which names normally start with the prefix “MPI” should also be accessible with the prefix “PMPI”. [18]

2.8.1 Profiling

The usual purpose of profiling is determining which sections need optimizing: the performance of the different parts of the program, how often functions are called, which functions are called and by whom, as well as memory and cache consumption. The main techniques for profiling are using code instrumentation (adding print-outs), instruction set simulation, operative system hooks and performance counters.

There are two types of profiling: invasive and non-invasive. Invasive profiling means modifying program code by inserting calls to functions that record data. This type of profiling is very precise. However, the overhead may be high depending on the efficiency of the inserted code. During invasive profiling only the application itself is profiled, not the complete system. During non-invasive profiling statistic sampling of the program is done. The sampling can be performed either using fixed time intervals, or using the performance counters available in the CPU. This

type of profiling has a low overhead and can profile the whole system including the kernel. However, only statistical data is produced.

[37]

2.8.2 Tracing

According to [25] a program trace lists the addresses of instructions executed and data referenced when a program runs. In this master thesis we will focus on packet tracing, which is a process by which one can verify the path of a packet through the layers to its destination [26]. Generally packet traces are produced in packet filters [27]. However, we will produce a packet trace on a higher layer – layer 5 (the layer in which MPI resides). The method blurs somewhat with invasive profiling described above. There is a number of profiling and tracing tools mentioned in [23]. I've been using a tracing library written by Torsten Hoefler called *liballprof*.

2.8.2.1 Liballprof

PMPI is the standard profiling interface of MPI. Being able to call standard MPI functions with both “MPI” and “PMPI” prefix allows one to write functions with “MPI” prefix that call the equivalent “PMPI” function. Functions with the “PMPI” prefix have the behavior of the standard functions, plus any other behavior one would like to add. This can be used for both capturing and later analyzing the performance data (central for this master thesis) and customizing MPI behavior, and this is exactly what has been done in liballprof library, which is a part of LogGOPSim (Section 3.1) tool chain.

Liballprof must be linked to the MPI application we would like to collect traces from. In this library the most important MPI functions have been “implemented”. A function's code typically does the following:

- write function name to the buffer
- write call time to the buffer
- call PMPI version of the function (i.e. the actual function) and store it's return value
- write all the function arguments to the buffer
- write the return time to the buffer
- return the stored return value

The buffer is written to file by a separate thread (for efficiency reasons). For every running process we get an output trace file containing information about all MPI function calls. Trace files reside in the /tmp directories of the nodes on which the application was run.

Chapter 3: Introduction to LogGOPSim and the IB Model

LogGOPSim and the IB Model are the two central simulation setups used in this master thesis for running simulations based on packet traces. Their integration will be presented in Chapter 4. LogGOPSim will be slightly changed during the integration – in this chapter it is the unchanged version that is presented, and we refer to it as "*original LogGOPSim*". In this chapter LogGOPSim and the IB Model will be presented separately, in a relatively detailed way. Much of the information in this chapter is based on the source code of the two setups.

3.1 LogGOPSim

LogGOPSim is a simulator program written by Torsten Hoefler and Timo Schneider from Indiana University. LogGOPSim has got its name from the LogGOPS model (Section 2.2).

LogGOPSim is a single cpu application. Its main goal is simulating short phases of MPI applications with up to 8 million processes. Simulating applications with reasonable number of messages, typically running for over five minutes, should be possible for up to 50.000 processes. LogGOPSim offers support for differentiating between eager and rendezvous sends (Section 2.7.5.1).

LogGOPSim may simulate the whole protocol stack with a high level of abstraction. The simulation of the layers below the Application Layer are based on the variables of the LogGOPS model (Section 2.2). The behavior of the application layer has to be reflected in the GOAL-schedule, on which the simulation is based (the description of GOAL is given later in this section). The abstraction level of LogGOPSim is high – as already mentioned it is based on the LogGOPS model, which only operates with link latency and per byte or per message processing and network injection delays. This leads to the excellent scalability of the simulator.

Internally LogGOPSim consists of two main parts: the parser reading input schedules and the core executing the simulation. The parser, besides reading input files, manages dependencies between events and execution order.

```
num_ranks 2
rank 0 {

       l1: calc 100 cpu 0
       l2: send 10b to 1 tag 0 cpu 0 nic 0
       l3: recv 10b from 1 tag 0 cpu 0 nic 0
       l2 requires l1
}
rank 1 {
       l1: calc 100 cpu 0
       l2: send 10b to 0 tag 0 cpu 0 nic 0
       l3: recv 10b from 0 tag 0 cpu 0 nic 0
       l2 requires l1
}
```

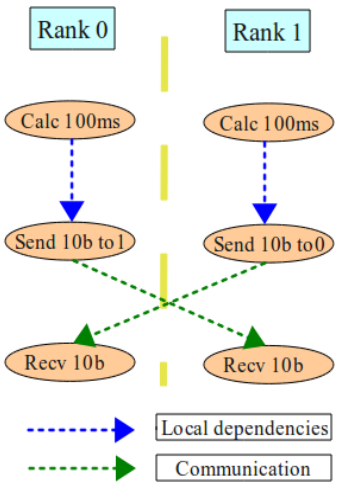

Figure 3.1.1 Example .goal schedules and the corresponding graph.

Inspired by [28]

A simulation schedule file is written in Group Operation Assembly Language (GOAL). GOAL is a language which can describe arbitrary parallel applications. There are three types of tasks defined in GOAL: send, receive and computation. Tasks are arranged in a directed acyclic graph. The dependencies between the tasks are the edges of the graph. Figure 3.1.1 shows a short scheme, written in GOAL, describing two processes first computing for 100 microseconds and then exchanging 10 bytes of data.

One text block between the curly brackets in Figure 3.1.1 is called a schedule. A parallel application with P processes would be represented by P GOAL schedules. The textual human readable schedule file consisting of schedules like the one in Figure 3.1.1 is converted into a binary GOAL schedule file for efficiency reasons. This binary file serves as input for LogGOPSim.

Naturally the .goal schedules can be written manually. However there is also a schedule generator (Schedgen) which is part of LogGOPSim tool chain. This schedule generator is capable of producing .goal schedules where the network traffic either follows some scheme (like the dissemination traffic pattern described in section 4.7.2) or where the traffic pattern is based on MPI traces.

3.1.1 The LogGOPSim Core

The simulation core is based on a single priority queue containing the executable events sorted by their earliest execution time. This queue is called the "Active Queue" (AQ). The events are added to the AQ by the parser. An event is added if it has no dependencies, or if all its dependencies are satisfied.

Figure 3.1.2 LogGOPSim Core Program Flow [8]

There are four types of executable events in LogGOPSim: sending a message, receiving a

message, “Message-on-flight” and local operation (processing for some time). The Message-on-flight event represents a message which is currently traveling through the network, i.e. departed from the source, but not yet arrived at the destination. Figure 3.1.2 illustrates schematically what is done in each of the four event handlers. The textual explanation comes below.

If a send event is retrieved from the AQ the following happens. First we check that local processing (o) and network (g) send resources are available. We have 3 counters per process: time until which the processor is busy, time until which the network sending resources are busy and time until which the network receiving resources are busy. If at least one of the two resources needed for a send operation is not available, the event's time is set to the time when both processing and network send resources get available and the event is reinserted into the AQ. If resources are available, we satisfy all the immediate dependencies on this event, so that the parser can insert the events depending on the start of this send into the AQ. An immediate dependency is a dependency that can be satisfied when an operation starts; this is done to model non blocking messages. Then the network sending and processing resources are charged (this is called “update o, g” in Figure 3.1.2). Charging the processing resource is done by setting the time when this resource gets available again to the time when the current send has started plus overhead per message plus overhead per byte (plus OS noise). Then we do the actual insertion of a message into the network layer (originally LogGOPSim comes along with a simulation of a network layer). We also add a new event representing the Message-on-flight into the AQ. The time for this event in the original LogGOPSim is set to the current time, which leads to this event being retrieved immediately after the current send event is handled. If the message being sent is an eager message (it is small enough for the eager protocol to be used), the normal dependencies are satisfied.

If a Message-on-flight event is retrieved the following happens in the original version of LogGOPSim. As we did with the send event, we check availability of local resources. In this case it's processing and network receiving resources. We also query the network layer for the earliest arrival time of the message. If at least one of the required resources is not available or the message has not arrived yet, we reinsert the Message-on-flight event into the active queue. The reinserted event's time is set to the latest of the three: time when the processing resource will be available, time when network receiving resource will be available and the message's earliest arrival time. If the message has arrived and the resources are available we first charge the receiving process' processing and network receiving resources in the same way we did during the send operation. Then we check if the message is in the receive queue. If it's not – we insert it into the unexpected queue (Section 2.7.5.1). If the message was not eager we can finally satisfy the dependencies for the sender process and set the sending and processing resource timers to current time for the sender. And no matter whether the message was eager or not, the dependencies for the receiver can be satisfied. Notice, that the matching of messages in the receive or unexpected queues happens using MPI semantics, i.e. the tuple <tag, source>.

Earlier we have mentioned the immediate dependencies which can be satisfied when an operation starts. The normal dependencies can be satisfied when an operation completes. If the receive event is popped from the AQ we first satisfy the immediate dependencies for the receiver. Then if the message is found in the unexpected queue we satisfy normal dependencies for the receiver. If the message was not eager we can also satisfy normal dependencies for the sender and set the sender's timers for sending network resources and processing resources to the current time. If the message is not found in the unexpected queue we post an entry into the receive

queue. Notice, how the actions taken in case the message is found / not found in the unexpected queue resemble the actions for the Message-on-flight event, for the cases found / not found in the receive queue.

The simplest event is local operation event which represents some local processing. If processing resources are available, we charge them and satisfy all the dependencies from this operation. If resources are not available we reinsert the event into the AQ so that it is retrieved when resources are available.

Let's look a bit closer at the already mentioned example where the two nodes first compute for 100ms, then exchange 10b of data. The default parameters for LogGOPSim are L=2500, o=1500 g=1000, G=6, O=0, S=65535. The active queue is initialized with 4 initial events which don't have any requirements: the two local operations and the two receives. Then we start popping events from the active queue. The four initial events are popped in the following order: receive, calculate, receive, calculate. In our case when a receive operation is popped we first satisfy the immediate requires (in this example it doesn't lead to the addition of any new events), then check weather the message already has arrived or not (is in unexpected queue or not). Naturally in our case the messages haven't arrived (the send operations of these messages haven't even been added to the active queue yet), so the receive requests are posted into the receive queue. When a calculation operation is retrieved, the local processing resources are charged with 100ms and the dependencies are satisfied. At this point the two send operations are inserted into the active queue (they depended on the calculation operations). Their time stamp is 100ms. Now the just inserted send operations are popped. Since the messages are small, the eager protocol is used and both immediate and normal dependencies are satisfied (no new events added). The two Message-on-flight events are added to the active queue during the handling of the send events. The processing per message (o=1500) and link latency (L=2500) are charged, so the time stamps for our two Message-on-flight events are 4100ms. Then these events are popped. The local processing per message (o=1500), network (G=6) and processing (O=0) overhead per byte of message are charged at the receiver. Since the receive posts are found in the receive queue, the messages can be considered received, and the simulation is done. One thing worth noticing concerns the 'G' parameter. It is charged for every byte of a message except the first byte (i.e. in our case 54ms are charged for 10b messages, instead of 60). So the total simulated time in our case is local calculations (100ms), plus link latency (2500ms), plus processing overhead per message which was charged twice: at the sender and at the receiver (1500ms+1500ms), plus network overhead per byte (54ms). All together 100+2500+1500+1500+54 = 5654ms.

3.2 Infiniband Simulation in Omnet++

In this master thesis the Infiniband simulation (IB model) in Omnet++ (Section 2.5) is a part of my simulation setup. This setup is used to run simulations based on packet traces (which the IB Model alone is not capable of). The traces are collected from MPI applications running on a real cluster where cluster nodes are connected using Infiniband. The model simulates an Infiniband (Section 2.6) network consisting of hosts and switches connected with links. Basically we are not talking about hosts here as we're not interested in what's going on in the application layer, but rather Host Channel Adapters (HCA). In Omnet++ modules can consist of several other modules. Both HCAs and switches in the IB model are compound modules. A HCA consists of input buffer, sink, virtual lane arbitrator, congestion control manager, output buffer and a traffic generator. A switch port compound module contains the same as a HCA, except for sink and

generator. A switch consists of several switch ports. The graphical representation of this is shown in Figure 3.2.1.

The running time of an Infiniband simulation in Omnet++ is proportional to the size of the simulated network and the length of simulation (i.e. simulated time in the end). How active the nodes are is also of importance for the runtime (the more active the nodes are – the more events). Generally we're talking about approximately 3 hours runtime per simulated second for a fat tree topology network of 8 HCAs and 6 switches on a single core of a Core2Duo T6600 @2.4GHz cpu. The memory consumption is around 50 megabytes for such a simulation. The memory consumption increases to about 250 megabytes for a fat tree topology network consisting of 648 HCAs and 54 switches (naturally the runtime increases too).

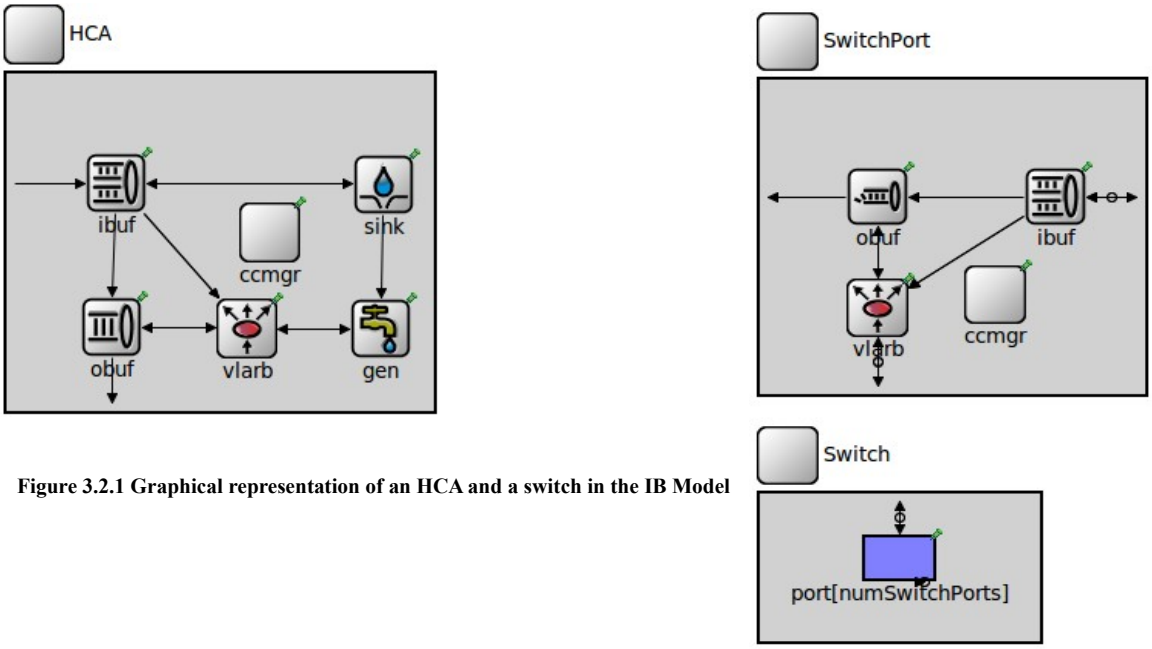

Figure 3.2.1 Graphical representation of an HCA and a switch in the IB Model

3.2.1 Input Buffer

This simple module is responsible for receiving packets from the output buffer on the other end of the link.

When a data packet arrives, two counters are changed: the counter representing free buffer space for a given Virtual Lane is decremented, and the counter of received flits is incremented. "*Flit*" stands for "flow control digit" and is the smallest unit flow control operates with [30].

The data packet is queued in the queue for the given output port. For switches the output port is determined using the forwarding table, for HCAs the traffic from the input buffer goes to the Sink.

If congestion control is enabled and we are in a switch we also update the congestion info for the given output port for the given Virtual Lane by sending the corresponding queue fill ratio and capacity to the congestion control manager.

The information about available buffer space per Virtual Lane is also provided to our output buffer, so that this number can be sent to the CA on the other side of the link in a flow control packet.

If this input buffer is part of a HCA, we send the received data packet (head of queue) to the virtual lane arbitrator (vlarb), if we are in a switch we send head of queue only if the previous one already has been passed on to an output buffer. The credits (buffer space) are freed when the vlarb actually sends our head of queue to the output buffer or sink, and we notify vlarb about the new head of queue.

If the received packet is a flow control packet we do the following: the flow control packet contains information about available buffer space of the CA on the other end of the link and the number of flits the other CA has sent us.

The number of sent flits is not necessarily equal to the number of received flits due to sending errors, so the number of received flits is adjusted. This adjusted number is provided to the output buffer, so that it can be contained in the flow control packets sent out.

The information about buffer state of the CA on the other side of the link is provided to our vlarb, so that it knows whether data can be sent out to this CA.

[29]

3.2.2 Output Buffer

The output buffer simple module can send out two types of packets: data packets and flow control packets. Sending a flow control packet is very simple: it just contains information about buffers in the input buffer (provided by the input buffer itself) and the number of flits sent out until the present moment (a local counter incremented for every data flit sent out). For every data packet sent the output buffer space is freed and (if enabled) the congestion control manager is notified about the new buffer state.

In real life pushing data into a link doesn't happen instantly, so our output buffer is also responsible for not sending packets out too often. There is a delay between each sending. The delay is calculated based on the link bandwidth (2.5, 5.0 or 10.0 Gbps), link width (4x, 8x or 12x) and the size of the data being sent.

[29]

3.2.3 Virtual Lane Arbitrator (vlarb)

This is the most complex of the three obligatory simple modules which all CAs have (ibuf, obuf, vlarb). The fact that a switch consists of several ports, each of which consists of an ibuf, obuf and vlarb makes things complicated. The vlarb functionality in switches needs to coordinate communication not only between the input and output buffers of a single port, but between the in- and output buffers of several ports. In an HCA the vlarb coordinates communication between the input (generator) and the only output buffer an HCA has or between input buffer and sink.

If congestion control is enabled, one or several congestion notification packets (CNP) may be waiting to be sent. They have the highest priority and are sent out first, unless we're in the middle of the sending of another packet (from a given input on a given VL).

After trying to send the CNP the following happens: to model the operation of the real life virtual lane arbitrator we have two tables (High and Low Priority) containing the limits of how much data may be sent out for each virtual lane. The algorithm for selecting which input and virtual

lane to use as current source is described later. When a limit from the first table is exhausted, we send one packet based on the limit from the other table, and then restore the limit from the first table, or in other words the High Priority Table specifies how many high priority packets may be sent before sending a low priority packet.

The choice of outgoing port and virtual lane is done only for the first flit of a packet, the subsequent flits of a packet are sent out based on this choice.

After the input port and VL choice is done, and if the arbitration is valid (see next paragraph), we deplete the limit of the chosen entry of the given table, notify the congestion control manager about this send, and actually do the send. The local copy of the counter for sent flits for the VL we send on is incremented. After the sending we also notify the ibuf from which we've just sent, so that this ibuf can increment the number of free credits for a certain VL and update head of queue.

An arbitration is valid if there is enough queue space in the output buffer to hold the chosen packet and if the input buffer is not busy with another port (the latter condition is only for switches).

The choice of the next VL and input port from which a packet is to be sent is based on a round robin algorithm. Notice, that there is no need to choose VL or bother with the two tables if only one data VL is present – we just select the input port in a round robin fashion. The algorithm is shown in Figure 3.2.3.

```
for each entry in the given limit table (an entry represents a VL)
        for each input port
                if enough credits are available on the opposite side of the link
                and the limit in the table entry is sufficient
                and we're not in the middle of sending another packet
        (should never occur though)
                and congestion control manager doesn't mind (injection delay expired)
                        then a packet from the chosen VL and port can be sent
```

Figure 3.2.3 The virtual lane arbitration algorithm.

The available input buffer space of the CA on the other side of the link is provided to the vlarb by the ibuf, which in turn gets this information in flow control packets. (Actually the calculation of this buffer space is not this simple, but equivalent to the method used in real Infiniband. [section 7.9.4.3 in 6])

[29]

3.2.4 Congestion Control Manager

The congestion control manager doesn't receive any packets or events from the other modules (only internal events for logging and gradually decrementing the index into the table of insertion delays). However it provides a set of functions called by other simple modules.

Whenever ibuf receives a data packet marked with BECN (Section 2.6.1.11), it calls “checkBECN” function of the congestion control manager. In this function we increase the index into the table of injection delays. The index is then gradually reduced back to the minimum value. There is a “race” between incrementing this index due to received BECNs and gradual decrements. The injection delays in the table grow towards the table's end; the number of values

in the table is at least 128.

If a data packet received by ibuf is not marked with a BECN, we call “checkFECN” function. In this function we perform a check whether the packet is marked with a FECN. If it is, we produce a CNP (congestion notification packet) and unless the special queue for these packets is full, we enqueue this CNP, to be dispatched later by the vlarb. The CNP will be sent back to the source of the data packet marked with the FECN.

Ibuf calls the canSend function before sending a data packet if congestion control is enabled. In this function we check whether the injection delay has expired.

The canSend function, called by the virtual lane arbitrator, checks that injection rate delay has already passed, which means that the congestion the control manager has no objections to sending the given packet.

Ibuf calls the updateCong function and checks whether the input buffer queue length exceeds the threshold. If the threshold is exceeded, the given port/VL is marked as congested, and the counter of congested ports for a given VL is incremented. The opposite is done whenever this function gets called and no congestion is detected.

Switch obuf calls the markFECN function for each outgoing packet. If there is congestion (input buffer queue for a given VL getting full), there is a chance that a packet will be marked with FECN.

[29][11]

3.2.5 Generator

As the name suggests this simple module is responsible for generating data packets, following a given pattern. Remember, that this is a simulation. So we're not talking about actual data packets with certain headers, but rather about events being pushed and popped from the priority queue; an event should carry information about what it represents, so an event representing a data packet should carry information about this data packet. The maximum size of a data packet is typically a multiple of 64; it was set to 2176 bytes when I was running simulations. This includes two obligatory headers: Local Routing Header (LRH) and Base Transport Header (BTH) which together are 20 bytes long. So the smallest packet size is 20 bytes. A packet is divided into 64 bytes large flits, if the the packet size is not a multiple of 64, the final flit can be smaller. If a large message consists of several packets, we mark each packet with the remaining number of bytes to send (for example for a 10.000 bytes message, 2156 bytes will fit into the first packet, and the remaining number of bytes will be set to 7844 bytes). A packet is also marked with the number of flits it consists of. And naturally we also set destination and source Local ID on a packet.

To be more precise, we operate on the flit level. When we need to send a message consisting of a certain amount of data, we actually generate a number of flits, each containing information about the message and the packet it is part of. A flit is sent when the head of queue (for this input) in vlarb is empty. When vlarb dispatches a flit it sends an acknowledgement to the generator, so that the generator can send the next flit (if one is available). A generator may also check if the head of queue in vlarb is empty by calling a function provided by vlarb. When we get a notification from vlarb about a dispatched flit, but have nothing to send for the moment, we do nothing fat that time, and only check the head of queue through the provided function when we get anything to

send. The same is also done in the very beginning, when sending the first flit of the first message. When we get a notification from vlarb and we actually have something to send, we just send it...

[29]

3.2.6 Sink

The sink simple module utilizes four internal events: push, pop, logging and hiccup. Push adds a flit to the FIFO queue, pop removes it. The periodic logging event is just for writing a log. PCI Express hiccups are simulated using hiccup events – these events cause the sink to alternate between two states: ON and OFF. During ON state any pop event is ignored, on transition to OFF state a new pop event is scheduled. In practice it means that the time is divided into periods when flits are consumed, and periods when flits are not consumed. Pop events are scheduled to arrive with a certain frequency depending on the simulated PCI Express width and transfer rate. So a sink doesn't just consume unlimited amounts of data, but has a certain efficiency. Consuming a flit basically means deleting it, recording some statistics, and rescheduling another pop message. The ibuf is notified when a flit is consumed, so that ibuf can update the head of queue and notify vlarb about that.

The external event received by the sink is another data flit to consume from the ibuf. This event causes the sink to schedule a pop event (unless one is scheduled already) with a certain delay.

Chapter 4 The Integration of LogGOPSim and IB Model

In this chapter a detailed description of the integration of LogGOPSim and the Infiniband Model in Omnet++ (sections 3.1 and 3.2) will be given. First we take a closer look at the design of the integration and it's evolution. Then results of several validation and efficiency tests are presented. A method for approximating the simulation time (i.e. the time it takes to run the simulation) will also be described.

Integrating the two simulators is the first main part of this master thesis, while evaluation of the setup is the second part. The integration is intended for running simulations based on packet traces (Section 2.8.2), using the IB Model in Omnet++ as the link and physical layers, and LogGOPSim as the upper layers.

4.1 Motivation for Integrating LogGOPSim and IB Model in Omnet++

In the Infiniband Model in Omnet++ the network traffic generated by the generator modules (Section 3.2.5), which are part of the end nodes (HCAs), is “artificial”. The synthetic traffic is characterized by the distribution of destinations, injection rate and message length. An example of distribution of destinations can be the *sphere of locality distribution*, where a node sends messages to nodes inside a sphere centered on the source node with high probability, and to other nodes with low probability. The injection rate often follows the exponential distribution, though uniform distribution within an interval or bursty traffic are also common. The message length can be fixed, or can be computed according to a normal distribution or a uniform distribution within an interval. [42 section 9.2]

This synthetic traffic does not necessarily reflect the real world network traffic coming from the upper layers and originating from the application layer. The original IB Model in Omnet++ is not capable of simulating any complex traffic patterns for several nodes, like the patterns of the MPI collective operations where sending of data messages may be triggered by reception of other data message(s).

On the other hand, LogGOPSim simulates the application layer and most importantly is capable of simulating the real MPI traffic given that the simulation is based on an MPI trace. In other words LogGOPSim takes care of any dependencies between data receptions/sendings which are found in the real life applications.

We want the best of the IB Model in Omnet++ and LogGOPSim: the former will act as link layer and the later will simulate the upper layers.

4.2 Approach to Integration

We need to connect the two simulators. One approach could be copy-pasting parts of LogGOPSim source code into the code of Infiniband simulation in Omnet++ and merging the two simulators into one. However I've chosen to leave them both more or less intact, which will make life easier if the newer versions of Infiniband simulation or LogGOPSim appear. My solution uses the interprocess communication in Linux (mostly Message Queues to be more specific). LogGOPSim and Omnet++ run as two separate processes communicating to each other with IPC.

Originally LogGOPSim comes with it's own module which is supposed to simulate the lower layers. My implementation is largely based on substituting the existing network-module of LogGOPSim by another one and adding an extra very special “Generator” node to the Infiniband network simulated in Omnet++ (though the name “Communicator” would probably more suitable for this Generator node and exclude any mix ups with the Generator simple module which is part of the HCA compound module).

Two major attempts for integration have been done. Both of them resulted in a working simulator. During the first attempt the two parts of the Integration were kept relatively separate, and LogGOPSim source code was not altered (except substituting one file containing the code for the network module by another). However, though functioning and being correct, the first approach turned out to be quite inefficient (too long simulation times), which was the motivation for making another attempt. The other attempt was more efficient (roughly 10% slower than the IB Model in Omnet++ running alone), however the LogGOPSim core had to be moderately changed, and the changes to the IB Model code were also more significant.

4.3 Overview of “integration”

Infiniband simulation in Omnet++ acts as link layer for LogGOPSim, that simulates the upper layers. Originally LogGOPSim expects it's “network” to provide two operations:

- Insertion of a new packet into the network (insert)
- Getting arrival time information about a previously inserted packet (query)

The integration, as already mentioned, is implemented using Linux interprocess communication. LogGOPSim and Omnet++ run as separate processes (Figure 4.3), and every time LogGOPSim wants to insert or query a packet, a message is sent between the two processes.

Every Omnet++ simulated network is expected to have a special module of class Generator which handles the interprocess communication. Furthermore every host module (HCA) is supposed to:

- Insert it's ID into a “database” during the initialization phase (Generator provides a function for this), so that the Generator knows who is present in the network and is able to send direct messages to all HCAs.
- Be able to accept direct control messages from the Generator. Control messages are insertion orders, containing packet destination, size and unique id.
- Provide information about arrived packages to the Generator, so that the Generator can forward this information to LogGOPSim. (this part is implemented differently in the two integration attempts).

We're using mostly Message Queues (man msgget). We also use a tiny slice of shared memory (man shmget) for Omnet++ process id (man getpid) retrieval by LogGOPSim. The only thing done by the network module in LogGOPSim is handling the interprocess communication with the Generator module in the Omnet++ simulation. There are three or two message queues in the first and second version of Integration respectively. Each message queue is dedicated to its own task to keep things as simple as possible (for example there is a queue dedicated solely to packet insertion messages, so there is no need to check what a retrieved message is).

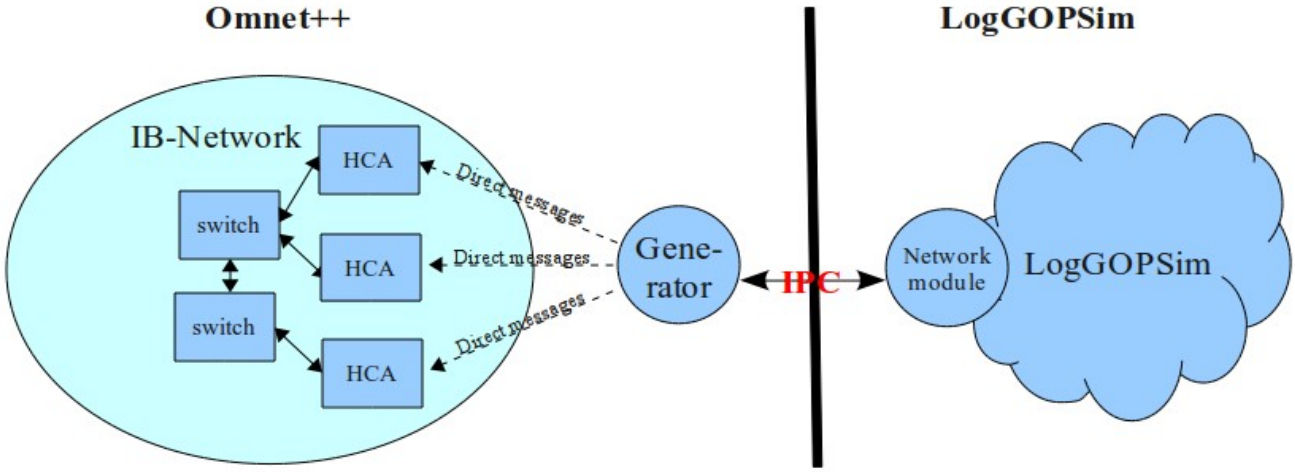

Figure 4.3 Schematic overview of Integration

4.4 Integration Using the Polling Mechanism

As already mentioned it was chosen to implement the integration as two processes running separately. So the main problem that had to be solved was inventing the co-simulation solution, i.e. something that would ensure the simulated times in both simulators to be advanced in parallel, or at least be equal at certain critical checkpoints. There are a few things that have to be considered in regard to the co-simulation solution:

1. The LogGOPSim originally communicates with its network layer through two functions: insert() and query().
    - If query() returns some moment in the future, the query is repeated at that moment.
2. It is the LogGOPSim that determines when data messages are to be sent or accepted.
3. There is no way to spool back the simulated time in Omnet++.

Considering points 2 and 3 above it is obvious, that if LogGOPSim wants to do a message insertion at time *T*, and Omnet++ has already passed this point of time – then the message insertion at time *T* would be impossible (we can't change the past). So we need to hold Omnet++ simulated time equal or smaller than LogGOPSim simulated time.

LogGOPSim always provides the current simulated time as one of the parameters when calling insert() or query(). The main idea of the synchronization solution is always advancing Omnet++ simulated time to the point of time provided by LogGOPSim. Since LogGOPSim simulates the events chronologically, we may be sure that every insert() or query() is provided with current simulated time *T* which is greater or equal to the simulated time provided earlier. So advancing the Omnet++ simulated time to LogGOPSim simulated time is always a safe operation, and will ensure that "message insertion into the past" situation never occurs.

It is a bit more complicated for message arrivals though. It is Omnet++ "half" of the integration that determines when a message arrives at destination. If LogGOPSim is planning to do a message insertion immediately after some message's arrival, then the arriving message is not supposed to arrive at a point of time smaller than LogGOPSim simulated time. If it does – LogGOPSim would have to insert the subsequent message into the past (which is something we

would like to avoid at this point). LogGOPSim gets the information about a message's arrival through the query() function. The first query() for a message is done immediately after message's insertion. When Omnet++ gets this query it is supposed to send back a reply with the arrival time of the message being queried. However, since the query arrives right after insertion, the message's travel through the network towards the destination has not been simulated yet. In other words there is no way Omnet++ can provide the message's actual arrival time without simulating it. Logically, to simulate the message's travel the simulated time needs to be advanced. However we're not allowed to advance the time past LogGOPSim simulated time provided in query(); and this time is currently equal to the message's insertion time... This looks like a dilemma, and finding solution to this was the most difficult challenge during the integration implementation.

Though the real world is continuous, the discrete-event simulations operate with discrete indivisible time slots. For example if such a time slot is 1 nanosecond, then no event can happen at time 9.5 nanoseconds after the simulation start – it has to be either 9 or 10 nanoseconds (the smallest time slot Omnet++ can operate with is picosecond). So, we are in Omnet++ and are supposed to provide LogGOPSim with message's arrival time without simulating it... We know, that if LogGOPSim gets some point of time in the future, then it would repeat the same query again at that point of time. So what we do is simply sending LogGOPSim the time point equal to LogGOPSim's current simulated time plus one minimal time slot. It is safe, because we know that if the message has not arrived yet, and no events can happen between the time slot boundaries, so the message will not arrive before that time. After sending LogGOPSim the time equal to current time plus one time slot, we do not advance Omnet++ simulated time yet – we wait for more inserts or queries from LogGOPSim, and advance the Omnet++ simulated time only up to the times provided with insertions or queries. We know that there will be a query at time equal to current plus one time slot. If the queried message still hasn't arrived we simply repeat the trick until it arrives – we keep replying to queries with time equal to current plus one time slot. The effectiveness of this approach will be discussed later, but it is a logically correct way of keeping both simulators synchronized and the simulation valid.

4.4.1 Message Flow During Packet Insertion

The message flow during packet insertion is shown schematically in Figure 4.4.1. In the insert function of LogGOPSim network layer we send an IPC message to the Generator (Omnet++ module responsible for IPC between Omnet++ and LogGOPSim). The message contains the current simulated time in LogGOPSim, the inserted packet's size, source, destination and id (handle). When receiving an insert message, the Generator sends a delayed direct control message to the host (Omnet++ module representing an HCA) which acts as source node for the packet. The control message is sent delayed and arrives at the source node at the “current time” received from LogGOPSim (this is always possible because the LogGOPSim current simulated time is always greater than or equal to the one in Omnet++). In this way Omnet++ and LogGOPSim simulation times are synchronized. The Generator module in Omnet++ schedules its next awakening to the last “current time” received from LogGOPSim. An “awakening” is when the Generator handles its own self-message. The Generator may accept several insert messages during one awakening (the simulated time is moved forward using internal self-messages; when receiving such message, the Generator awakens, checks the message queues for new messages, then reschedules the self message to some moment of time greater than or equal

to the current simulated time). The Omnet++ module representing an HCA accepts the control message from the Generator and sends a message with the given size and id to the given destination. When the destination host accepts the message, it creates an entry in the arrival time "database" (message id → arrival time).

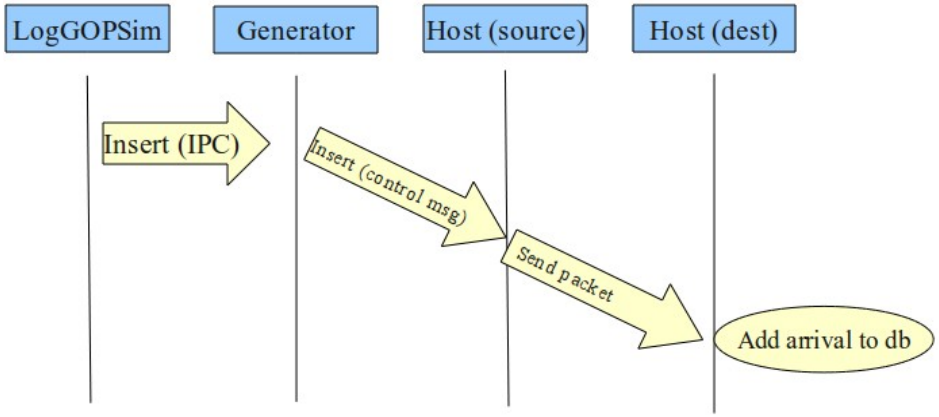

Figure 4.4.1 Schematic overview of message flow during message insertion

4.4.2 Message Flow During Query

The message flow during query is shown schematically in Figure 4.4.2. The aim of querying is retrieving the information about message arrival, so that LogGOPSim can start simulating whatever is supposed to happen after the arrival of the queried message. In the query function of LogGOPSim network module we first wait for both the query message queue and the insert message queue to be emptied by the Generator (sometimes one or both of these is not needed when a queue already is empty.) This emptying is important for synchronization reasons. Then we send the query message containing the current time and packet handle. After that we wait for the reply. The query function is blocking – it doesn't return until the reply from the Generator is received.

When the Generator receives a query message, all it does is setting a certain variable indicating that the query is not finished, and schedules the next awakening time to the LogGOPSim "current time" received with the query message. Notice that LogGOPSim will be blocked in the query function until it actually gets the reply. When Generator awakens again it resumes this query – no new messages could arrive because LogGOPSim was blocked all the time, and both message queues were empty by query start. Now there are two possible ways: the packet may have arrived already or it may not have arrived yet. In the first case we just send it's arrival time to LogGOPSim. In the second case we send back the "current time" incremented by 1 (a moment in the future that is), which makes LogGOPSim send the same query again after the minimum time slice has passed. From this moment it will "flood" the Generator with query messages until the packet actually arrives and the Generator confirms the arrival. (This is the feature which makes the Integration ineffective – see Section 4.8 for details.)

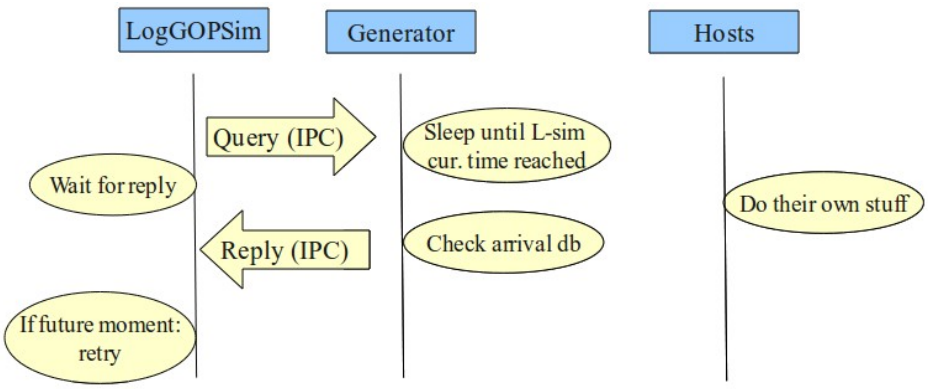

Figure 4.4.2 Schematic overview of message flow during query

4.4.3 Optimization

Efficiency testing (Section 4.8) has revealed that the setup described above is not effective. With the parameters related to the network injection rates and internal processing times in LogGOPSim set to zero, LogGOPSim sends a query immediately after packet insertion. This means that the reply from Omnet++ side contains some future time (time one minimum time unit larger than current time) and LogGOPSim has to send another query when this time comes... In other words one query is sent per cycle per message. For example if one cycle is equal to one nanosecond, and we are simulating the flight of 10 packets, each of which takes 10 nanoseconds to travel through the simulated network, we end up sending 100 queries. In real simulations we have many more messages, with possibly much larger travel time, and the minimal time units (precision) may be smaller than a nanosecond. This leads to relatively large number of queries per packet, which is ineffective.

One possible way of optimizing the integration was trying to estimate the earliest arrival time of a message at insertion, so that queries don't come before this time. If the estimate is relatively precise, lets say half of the real message traveling time the number of queries is halved. The safe way of estimating the travel time is taking the combined link latencies along the packet's path. However, it turned out that the real traveling time of a message was several times larger than the combined link latency time. This means that we win very little by this optimization. Multiplying this estimated minimal traveling time by some factor increases the benefit, sometimes up to 30-40%. However for different simulations this factor is different and the only way to find the optimal factor is by experimenting (first running the simulation with factor=1, then running it several more times and increasing the factor until the results are wrong) which is not practical at all. Even then, if a network is congested, the real traveling time can be much larger than the estimated one. Using too high factor will mean that the estimated time is larger than the actual arrival time which causes LogGOPSim to do the insertion(s) depending on the arrival at a later simulation time (which is wrong).

Notice that this solution requires building a “routing table” containing the number of hops between the different destinations at startup. A solution more efficient for small simulations would be calculating the number of hops separately for each packet, instead of building a “routing table” most of which may be unused. However I chose a “routing table” solution which implies fixed overhead per simulation. The overhead depends on the size of network simulated –

the larger the network, the larger “routing table” we get, the more time it takes to build it.

4.5 Integration Without Polling

Due to inefficiency of the integration setup involving polling (queries), some improvements had to be done. These involved changes to the functionality of LogGOPSim related to polling.

In LogGOPSim during message insertion (send operation) an event representing message on the way is inserted into the active queue of events (a priority queue upon which the simulation in LogGOPSim is based). Originally the time when this event is scheduled to occur is the message's insertion time plus some processing overhead. When this event occurs a query is to be sent to the network. The query is supposed to retrieve the earliest arrival time of the message, so that when this time comes, the query can be done again... until we discover that the message actually has arrived (when the retrieved time is equal to current time). Then the events depending on this message's arrival have a chance to start.

This setup has been changed because the network (the IB Model) has no chance of knowing when the message arrives, and replying with current Omnet++ simulation time plus one time unit (in other words polling every cycle for every message currently in flight) is simply too slow and ineffective, so this mechanism needs to be improved or bypassed.

The polling mechanism has been bypassed in the second version of Integration. When a message is inserted into the network, we set the time for the Message-on-flight event to the maximum value (for 64-bits unsigned integer). Eventually the active event queue contains just such events set to max time. When an event with max time is retrieved from the active queue (i.e. no more message insertions scheduled at the current moment) we send a notification to the IB Model that no more inserts are coming (so that the IB Model can interrupt its waiting for inserts and start simulating) and then we start waiting for a signal (interprocess message) from the IB Model (our network) indicating arrival of some packet. Retrieving a Message-on-flight event with maximum time from the active queue in LogGOPSim also means that all the event chains are blocked (all the messages are on the way and have not arrived yet). When a message finally arrives at its destination in the network simulated by the IB Model the signal is sent to LogGOPSim. However, the Message-on-flight event previously retrieved from the active queue in LogGOPSim not necessarily represents the message which has just arrived. If it doesn't we push the previously retrieved Message-on-flight event back into the active queue (with a greater AQ-insertion time stamp so that it is added last in the priority queue), and start popping the events from the active queue (and pushing them back into the end) until we find the one we are after, i.e. the one representing the just arrived message. We set it's time to the arrival time, check that there are available system resources on the receiving node, and add the events depending on the arrival of the just arrived message to the active queue (in other words we proceed handling this Message-on-flight in the usual way).

Lets look at an example to make things clearer. Suppose we have a network of two hosts. In the beginning each host sends a message to the other one. When receiving a message from the other host, a reply is sent back. So there are two messages on the way most of the time. LogGOPSim does one message insertion for each node, and adds two events representing messages on the way to the active queue. The time for which these events are scheduled is the maximum time. Then one of these events is popped from the active queue, we notify the IB Model that we're done inserting and we start waiting for the signal from the IB Model. The signal arrives, but it

tells us about the arrival of the other message (50% chance for that), not the one which' corresponding event just got retrieved from the active queue. We push the event into the end of the queue, and pop the next one. Since there were just two such events, this is the one representing the just arrived message, otherwise we would push it too, and pop events until we find the one we're after. So the message has arrived, and the reply can be sent back. An insertion is done. Similar sequence happens for the second initial message. The two reply messages are on the way and there are two events scheduled in LogGOPSim for maximum time representing these messages. We pop the first of them, notify the IB Model that no more inserts are coming for the moment, and wait for signal from the IB Model about an arrival, if needed we find the event representing the just arrived message... and one of the hosts is done. Shortly after the same happens for the other host, and the simulation is finished.

From the IB Model point of view the following is happening. Initially we wait for inserts. Two inserts arrive followed by the notification that there are no more inserts for the moment. We simulate the travel of these two packets. When the first of them arrives we send a signal to LogGOPSim and wait for inserts (we wait up to several thousand cycles). Insertion of the reply for the just arrived packet comes. Notification “no more inserts” comes. The second of the initial packets arrives and we signal LogGOPSim about that, wait for inserts, get the insertion of the reply followed by the notification “no more inserts”. When the first reply arrives we signal LogGOPSim about it, simulate the arrival of the second reply, send signal to LogGOPSim, wait for more inserts (none are coming) for a few thousand cycles and the simulation is done.

Figure 4.5.1 illustrates the above example with two hosts each sending one message and a reply. This illustrates how setup works in **real** time.

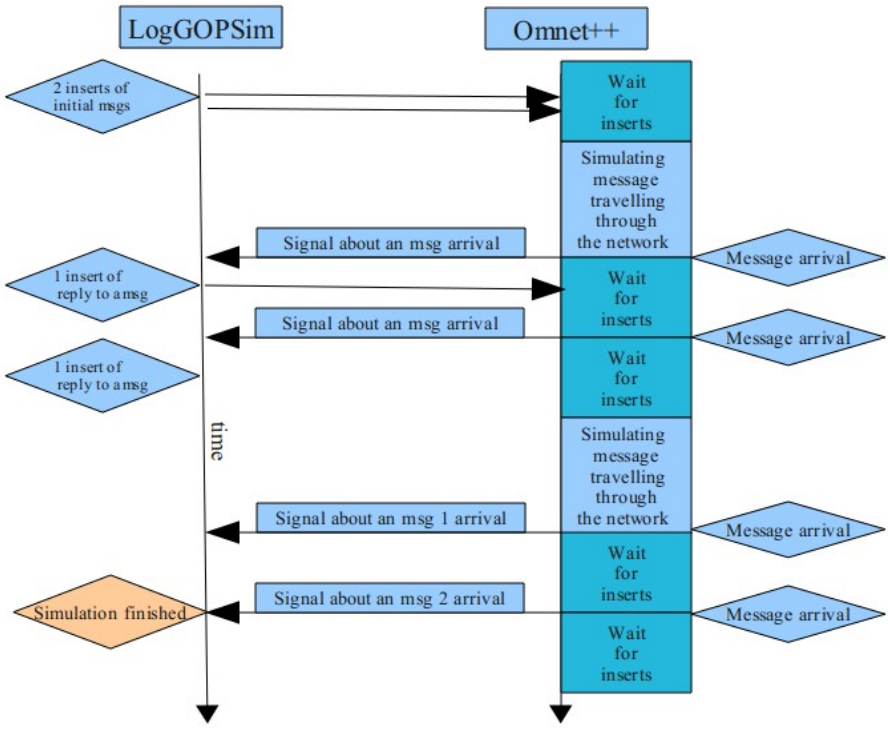

Figure 4.5.1 An example of message flow in the second version of Integration from the real time point of view

Figure 4.5.2 illustrates the event flow for a single message insertion and traveling. The

simulated time increases towards the bottom of the figure. The event labels increase with the real time.

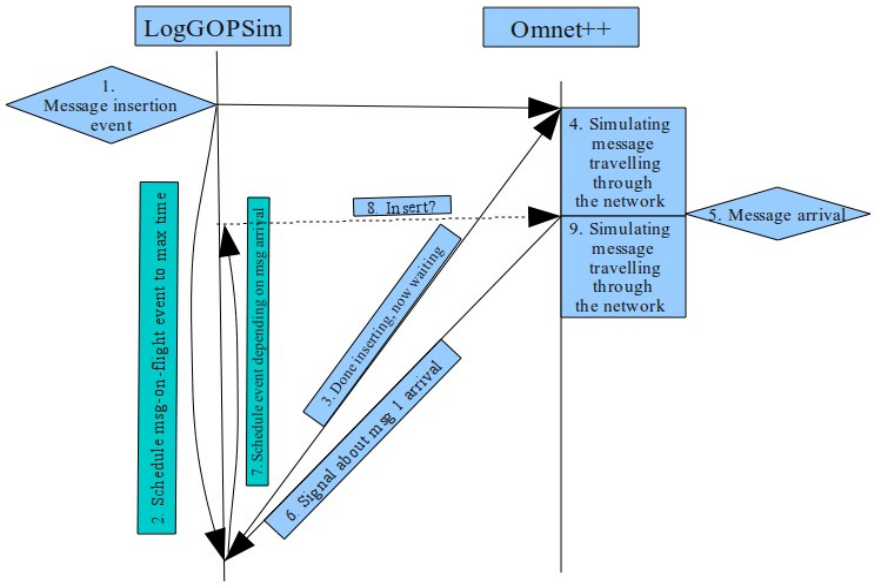

Figure 4.5.2 The event flow for a single packet from the simulated time point of view

4.6 Addressing

The traces on which the simulations may be based are collected from an MPI application running on a real Infiniband subnet. Each element of this subnet is identified with a Local ID (LID). Every MPI process is identified by a number, called “rank”. In the Infiniband Model in Omnet++ each node also has a LID, and the packets heading towards this node should have the node's LID in the “destination field of the packet header”. These LIDs are also used in the forwarding tables for the switches in the model. The LIDs in the IB Model start from 1 and increase contiguously onwards. Notice, that LIDs in the IB Model are not equal to the real LIDs in the cluster (it is not needed).

All modules (both simple and compound) in Omnet++ are identified with an Omnet++ module ID.

The trace files produced by the liballprof library are being numbered from 0 and onwards. The node with lowest MPI rank gets the lowest trace number. The numbers identifying ranks in the .goal schedule correspond to the trace numbers.

There are three different IDs which are important for us: the node ID in LogGOPSim (which is an MPI rank number), the module ID in Omnet++ and the LID of a simulated Infiniband node. In the Omnet++ module (“Generator”) responsible for the interprocess communication between the IB Model and LogGOPSim there is a data-structure holding triplets of these IDs. Every gen submodule of a HCA during initialization must call a function provided by the Generator and provide these three IDs. For this it was necessary to introduce a new parameter for the gen simple module. This parameter, called “rank”, contains the MPI rank of the process which was running on the real subnet node simulated by the HCA the gen is part of.

When the Generator gets an order from LogGOPSim stating that rank 0 should send some data to

rank 1, the data-structure containing triplets of Omnet++ module ID, LID and MPI rank is accessed to get the Omnet++ module ID of the source node (rank 0) and LID of the destination node (rank 1). Then a direct message is sent to the source (this is what Omnet++ module ID is needed for) commanding it to send the required amount of data to the destination LID. (Section 4.4.1)

The gen simple modules receive information about the rank of the MPI process which was run on the corresponding HCA through the “rank” parameter. For example let's say that the MPI application has been run on a real Infiniband subnet, and consisted of 2 processes. In such case the person running the simulation should make sure that the gen simple modules of the two HCAs participating get respectively rank=0 and rank=1 parameters provided in the .ini file. Notice, that the person should be completely aware of which ranks run on which physical nodes in the real topology (using --hostfile option for mpirun (man mpirun) can be helpful not to lose control here).

4.7 Verification / Validation

Two series of 5 verification tests have been run. The difference between these two sets lies in the LogGOPSim parameters. Within each set four different topologies of Infiniband networks were used in the IB Model. In addition the tests were run on the original LogGOPSim alone with the original network module. The goal for running these verification tests is verifying that the Integration is functional and sane. The tests are fairly simple, so one can predict the approximate final simulated time. Something may be wrong with the setup for example if the results achieved by running the tests diverge a lot from the expected results, or if all the results are equal, or if one of the simulated times is an order of magnitude higher than the others, etc.

During a simulation the nodes are sending and receiving data. In our case the pattern for the traffic is determined by a synthetic simulation schedule. The final simulated time for a node is the time when the node has completed all operations that were scheduled for this node. The longest final simulated time among all nodes is the final simulated time for the whole simulation – there will be no simulated data-traffic after this time (only the flow control packets in the Infiniband network).

4.7.1 Verification Test Topologies

As already mentioned, four network topologies were used under the verification tests. These topologies have slightly different properties when it comes to the average number of hops and throughput. The variability of the network topology properties is expected to cause the different simulated times for the tests with the same traffic pattern run on different networks. The traffic pattern will be described in the next section.

- Topology H8_S1 (Figure 4.7.1.1) consists of 8 hosts and one switch. This “star” topology network doesn't contain any bottlenecks since the switch is nonblocking. There is constant number of hops (2 hops) between any two hosts. This network is expected to have excellent performance when it comes to throughput and latency.

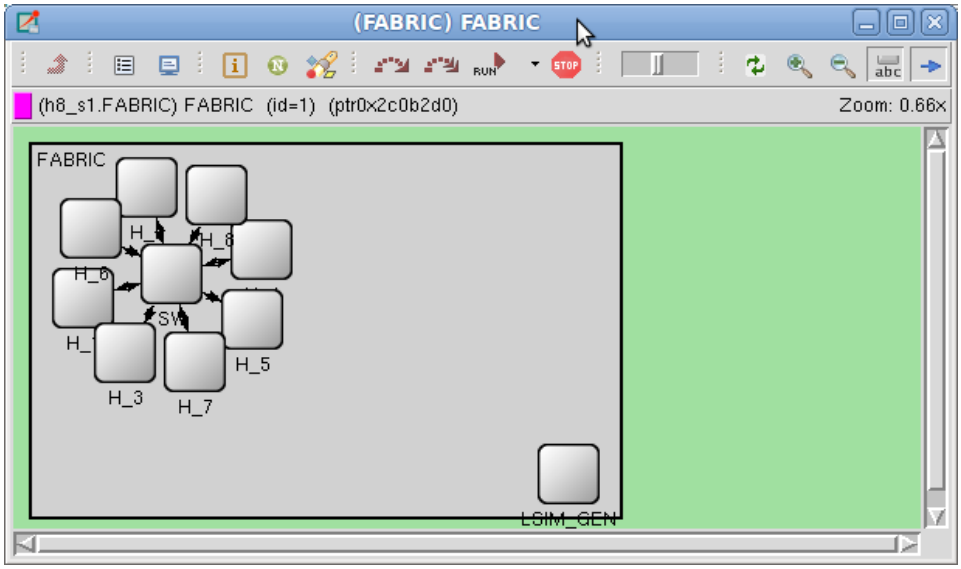

Figure 4.7.1.1 The H8_S1 topology.

- Topology H8_S2 (Figure 4.7.1.2) consists of 8 hosts and two switches. The hosts and switches form two stars (4 hosts and one switch in each) connected with a single link. This link will be the bottleneck when data is sent between hosts belonging to the different stars. This network is expected to have limited throughput for random traffic and longer average latency than H8_S1.

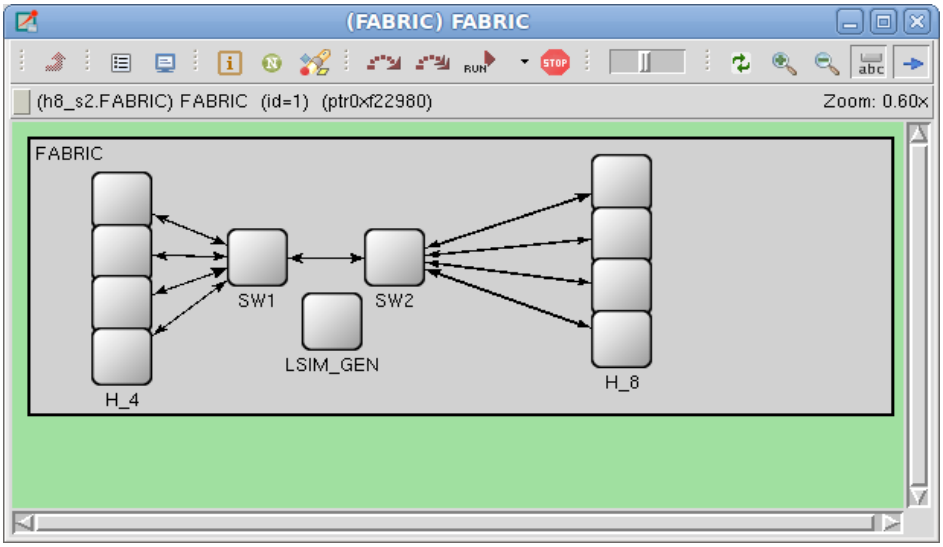

Figure 4.7.1.2 The H8_S2 topology.

- Topology H8_S4 (Figure 4.7.1.3) consisting of 8 hosts and 4 switches is another topology with a bottleneck, however the bottleneck here is wider than for H8_S2. It consists of 4 stars (2 hosts and 1 switch in each) interconnected with each other. The throughput is expected to be better than for H8_S2 while the average latency is longer.

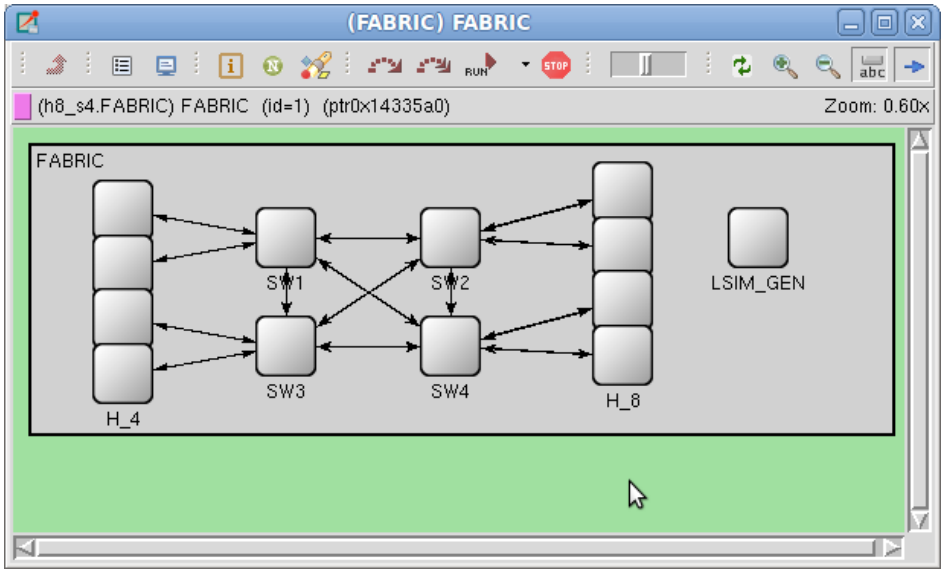

Figure 4.7.1.3 The H8_S4 topology.

- Topology H8_S6 (Figure 4.7.1.4) is a full bisection bandwidth topology consisting of 8 hosts and 6 switches. There are no bottlenecks here, while the latency is supposed to be the highest of all the listed topologies.

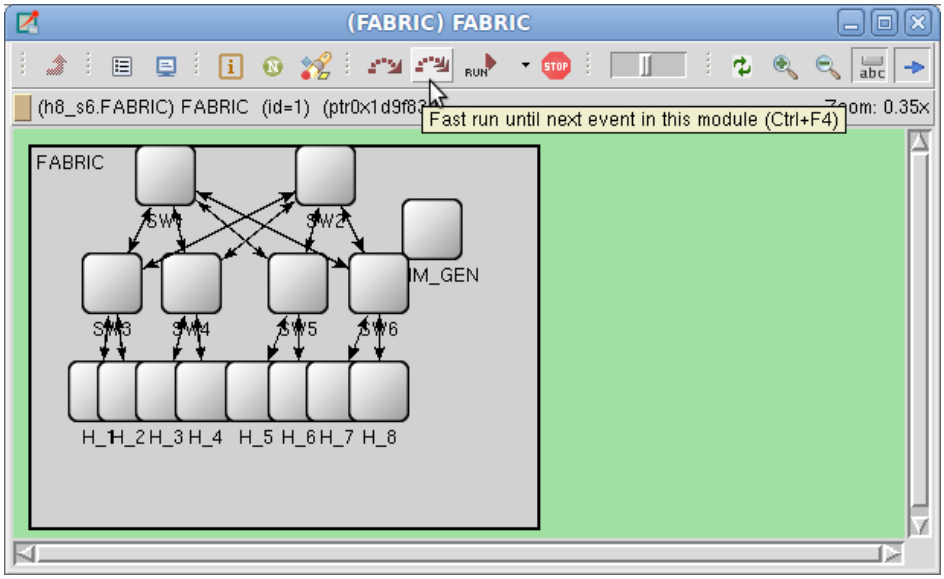

Figure 4.7.1.4 The H8_S6 topology.

4.7.2 The Test Traffic Pattern

Dissemination is a traffic pattern used in all-to-all data exchanges, barriers and allreduce collective operations. The sending distance is growing exponentially. This pattern is suitable for our verification tests because all the nodes are participating equally and are all sending and receiving messages relatively synchronously. (This is not the case for example for the binomial tree pattern [8], where the choice of the tree root may change the final simulated time for some topologies.)

In our case the dissemination among 8 nodes took place. This pattern is illustrated in Figure 4.7.2.1. Initially each node sends out a message to the node with the sequence number (rank) 1 lager than it's own. For example node 0 sends to node 1, node 1 sends to node 2, ... and node 7 sends the initial message to node 0. When a message is received, it is sent further to the node which is double as far away as the destination of the previous sending (i.e. node 0 sends the initial message to node 1, then after receiving the message from node 7, it sends a message to node 2, and then the final message to to node 4). So the number of messages each node sends is equal to base 2 logarithm of the number of nodes. The message size is always 180 bytes.

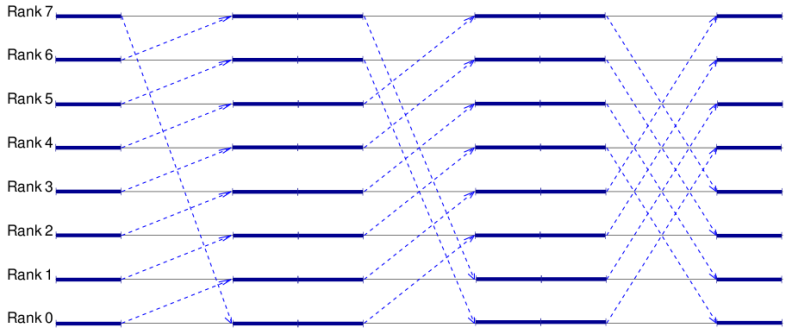

Figure 4.7.2.1 The dissemination traffic pattern [8]

When running a test with relatively few participating nodes, LogGOPSim prints out the simulated finishing times for each node. For each of the verification tests LogGOPSim has printed out 8 simulated finishing times.

The first set of tests has been run with LogGOPSim parameters o=50,000, g=100,000, G=6000, L=0. All the parameters and times are here measured in picoseconds. The summary of the first set of tests is presented in the Table 4.7.2.1 below. The summary consists of the minimum and maximum out of the 8 times, and the standard deviation.

Test topology	Max time value	Min time value	Standard deviation
LogGOPSim (no topology)	3522000	3522000	0
H8_S1	4293200	4293200	0
H8_S2	4667100	4378750	114575
H8_S4	4577250	4473350	55536
H8_S6	4860850	4667100	103563

Table 4.7.2.1 The summary of the first set of verification tests with LogGOPSim parameters o=50,000, g=100,000, G=6000, L=0. The units are picoseconds.

The second set of similar tests has been run with LogGOPSim parameters equal to 0. The results are presented in Table 4.7.2.2 below.

Test topology	Max time value	Min time value	Standard deviation
LogGOPSim (no topology)	0	0	0
H8_S1	723100	723100	0
H8_S2	1098500	780600	125436
H8_S4	943100	843100	53452
H8_S6	1261850	1105600	83519

Table 4.7.2.2 The summary of the first set of verification tests with LogGOPSim parameters o=0, g=0, G=0, L=0. The units are picoseconds.

4.7.3 Summary

If the traffic pattern was random, the average number of hops (traversed links to reach the destination) for packets would be:

- 2 for H8_S1
- 2.57 for H8_S2
- 2.85 for H8_S4
- 3.7 for H8_S6

We can clearly see that the reduction of average number of hops is the most important factor in our case for the overall simulated times because the simulated time is lowest for H8_S1, highest for H8_S6 and somewhere in-between for H8_S2 and H8_S4.

The bottle necks in the network proved to be less important for the simulation time for this particular traffic pattern because the topology with the relatively high throughput (traffic capacity) H8_S6 turned out to be the least effective. In the dissemination traffic pattern the number of packets currently moving through the network is always smaller or equal to the number of end nodes. If the pattern was different however, for example if the the nodes of the "left star" of H8_S2 network flooded the network with packets heading to the right star (so that all the traffic would have to go through the single link in the middle) the picture would probably be different.

As we also can see reduction of processing times on the application layer (o and g parameters for LogGOPSim) also causes overall reduction of simulation times. Verifying this was the reason for using two sets of tests. This is quite natural because the longer the processing in the application layer – the longer time goes between packet insertions into the network and the longer simulation time it takes for the whole thing to get completed.

4.8 Efficiency Testing

We've got five setups to test:

- Pure LogGOPSim simulation
- Pure IB Model simulation
- LogGOPSim & IB Model integration – the original setup
- LogGOPSim & IB Model integration – the original optimized setup (estimating arrival time at insertion)
- The new LogGOPSim & IB Model integration setup

The original setup can be run with several different precision levels (minimal time slot in the simulation). It makes sense to run with precision levels of 1 nanosecond, 100 picoseconds, 10 picoseconds and 1 picosecond. Higher precision means slower running. The original optimized setup can also be run with several optimization levels (factor by which the combined link latency time is multiplied).

The topology of the IB network used for the efficiency tests is called M9 and is similar to H8_S6 topology used earlier (full bisection bandwidth). M9 however is a lot larger – switches have 36 ports each, there are 18 switches on depth 0 of the "tree", 36 switches on depth 1, and there are 648 host channel adapters (nodes) on depth 2. The traffic pattern is again dissemination. LogGOPSim was run with L, g, G and o parameters equal to zero, which means zero processing

overhead at the application layer.

Hardware: Intel Core2Duo T6600 @2.4GHz, 4GB memory

Memory usage was approximately the same under all the tests. LogGOPSim used around 2.5MB of memory compared to 230-240MB used by the IB Model simulation. Such memory consumption by Omnet++ is mainly due to network size – with smaller networks of 8 nodes and 1-4 of switches (as in Section 4.7.1) the memory consumption was around 50MB.

Running the pure LogGOPSim simulation was very simple and not surprisingly it was the fastest setup. The running time printed out by LogGOPSim is rounded to the nearest second. The dissemination among 648 nodes takes 0 seconds to run (instantly from the human point of view).

I've done three tests with pure IB Model simulation and the total running times under these three runs were: 15.7s, 15.53s and 15.9s. Of these respectively 0.26s, 0.21s and 0.25s were used by the Generator module (in this case driven not by LogGOPSim, but by a simple algorithm and responsible only for commanding the nodes to send data; no IPC communication). As we can see in this case the Generator causes just about 1 or 2% of overhead which is considerably less than in the original IB Model & LogGOPSim integration case where Generator overhead tends to be 60-80% of the running time. The reason for such difference between Generator overheads in the two setups is the following. In the pure IB Model setup when a node receives a data packet it acknowledges the Generator, and the Generator commands it to send the next packet (unless the node is finished). This is a nearly perfect schedule without Generator performing any extra work cycles (previously called “awakenings”). In the IB Model & LogGOPSim integration setup it is not the Generator, but LogGOPSim that decides when the packet is to be sent. LogGOPSim sends query messages to the Generator when it wants to know whether a packet has arrived or not. The Generator has access to this information only after packet arrival, so, when not possessing the needed information, it simply replies to LogGOPSim with packet arrival time in the future, so that LogGOPSim sends another query later. This means lots of extra cycles for the Generator. In addition the Generator performs lots of idle cycles – a cycle when it doesn't receive an IPC message from LogGOPSim. All this causes a dramatic increase in Generator running time.

Running times of not optimized original Omnet++ & LogGOPSim integration vary with precision levels (simulation results are also slightly different between nanosecond precision and higher precision). With nanosecond precision the integration takes about 35.8 seconds to run, out of which about 16 seconds were spent in the Generator; with 100 picosecond precision – about 200 seconds; with 10 picosecond precision – about 1900 seconds (~half an hour). I haven't run this test with picosecond precision, but I expect the running time to be approximately 10 times larger than for the 10 picosecond precision test. This means that at higher precision levels it is the Generator that consumes most of the simulation time.

Running times of optimized original IB Model & LogGOPSim integration vary with both precision levels and optimization levels (factor by which we multiply the estimated packet arrival time). At optimization level 1 (estimated arrival time = combined link latency on the packet's path) it takes 35.6, 195 and 1775 seconds to run the simulation with 1000, 100 and 10 picosecond respectively. At optimization level 2 (estimated arrival time = combined link latency * 2) the times are 34.4, 181 and 1600 seconds. At level 3: 33.4, 175 and 1488 seconds. At level 4: 32.2, 163 and 1475 seconds. The summary of these results is in Table 4.8 below. Notice that the total running times are not precise for the old setup. There is no way for the Generator to know

when LogGOPSim is done, so the Generator performs some idle cycles in the end (after LogGOPSim is done) without doing anything.

	not optimized	**x1**	**x2**	**x3**	**x4**
1000 picoseconds	35.8	35.6	34.4	33.4	32.2
100 picoseconds	215	195	181	175	163
10 picoseconds	1900	1775	1600	1488	1475

Table 4.8 Simulation times for different precision and optimization levels

The same simulation takes just 16-17 seconds on the new LogGOPSim & IB Model integration setup which is just about 10% longer than the pure Omnet++ simulation.

To summarize what is said above, the Generator simple module consumes about half of the simulation time at low precision levels, and most of the simulation time at higher precision levels if we use polling in the integration. In the improved version without polling the Generator uses only about 10% of the simulation time and the results of the simulation have the highest (picosecond) precision. Running a pure IB Model simulation is most effective, with the Generator consuming only 1-2% of the simulation time. However the pure IB Model approach is not general in the sense that the different tests have to be programmed separately, instead of playing back the MPI traces.

4.8.1 Estimating The Simulation Time

It may be of interest to find out how the simulation time depends on the simulated time and the number of nodes being simulated. In addition this information can be used to get a rough estimate of the simulation time, though it greatly depends on one's hardware. Three series of tests have been run on four topologies of varying sizes.

The topologies are similar to the previously mentioned fat tree topologies H8_S6 and M9. Here is the summary of the four topologies used:

- H128_S24 consists of 128 HCAs and 24 switches. Each switch has 16 ports. There are 8 switches on depth 0, and 16 switches on depth 1.
- H242_S33 consists of 242 HCAs and 33 switches. Each switch has 22 ports. There are 11 switches on depth 0, and 22 switches on depth 1.
- H392_S42 consists of 392 HCAs and 42 switches. Each switch has 28 ports. There are 14 switches on depth 0, and 28 switches on depth 1.
- H512_S48 consists of 512 HCAs and 48 switches. Each switch has 32 ports. There are 16 switches on depth 0, and 32 switches on depth 1.

The traffic pattern is very simple in all tests. Each node does a certain number of sends (in each test this number is the same for all nodes). The message sizes are always the same – 1000 bytes. All the sends except the initial one happen after doing a receive. The destination's rank is always equal to the source's rank + 1. The results are presented in Table 4.8.1 below.

Topology and conditions	Simulation time [s]	Simulated final time [ps]	Number of events	# events w/o sends
128 HCAs, 24 switches, 15 sends a 1000b	5.25	10147000	1239827	701313
242 HCAs, 33 switches, 15 sends a 1000b	10.55	10140750	2349118	1325193
392 HCAs, 42 switches, 15 sends a 1000b	18.11	10140750	3741826	2101513
512 HCAs, 48 switches, 15 sends a 1000b	24.81	9949350	4904769	2744833

Topology and conditions	Simulation time [s]	Simulated final time [ps]	Number of events	# events w/o sends
128 HCAs, 24 switches,30 sends a 1000b	10.83	20247000	2551377	1441665
242 HCAs, 33 switches, 30 sends a 1000b	21.79	20086850	4788938	2701447
392 HCAs, 42 switches, 30 sends a 1000b	38.3	20086850	7806059	4375897
512 HCAs, 48 switches, 30 sends a 1000b	50.7	19897000	10150826	5668353

Topology and conditions	Simulation time [s]	Simulated final time [ps]	Number of events	# events w/o sends
128 HCAs, 24 switches, 45 sends a 1000b	16.74	30355600	3874778	2185729
242 HCAs, 33 switches, 45 sends a 1000b	33.06	30149350	7234240	4080121
392 HCAs, 42 switches, 45 sends a 1000b	56.82	30034500	11785702	6614217
512 HCAs, 48 switches, 45 sends a 1000b	78.08	29868100	15374041	8598529

Table 4.8.1 Results from running the efficiency tests to determine how the simulation time depends on the number of nodes being simulated.

In the first column of the table above we see number of HCAs and switches in a topology, and the number and size of messages each node sends during the simulation. In the second column stands the number of seconds it took to run the simulation. In the third column there is the simulated time in the end of simulation; notice that due to the traffic pattern these times are supposed to be approximately the same for the same number of messages being sent by each node, no matter the size of topology. In the fourth column the number of events in Omnet++ during the simulation is presented. And finally, in the right hand side column there is the number of events in Omnet++ if we simulate the same network, without HCAs sending anything, until we reach the same simulated time (these events are mostly due to flow control in the IB Model and internal events within HCAs and switches, like on/off events in HCA sinks.)
The simulated networks consist not only of HCAs, but also of switches which have several ports each. Increasing the number of HCAs normally leads to increasing the number of switches or switch ports, so we are not talking here about any precise dependency between the number of HCAs in the network and the simulation time – the number of switches influences the simulation time too. However the general tendency is clear. If the increase in the number of switches and switch ports is proportional to the increase in number of HCAs, then the number of events also increases proportionally, which in turn leads to the corresponding simulation time increase. In other words the simulation time increase is approximately linear and proportional to the increase in the number of events. (We are not interested in the special cases, when the number of nodes gets so large, that we run out of physical memory and the swap file gets used, which will greatly degrade the performance.)

Logically enough, the simulated time is proportional to the number of sends being performed by each node. The simulated time for the simulation with 45 sends is triple as large as the simulated time for the simulation with 15 sends. We can also see that the simulation time for the tests with 45 sends are approximately triple as large the simulation times for the corresponding tests with 15 sends.

The number of events for the 15-sends simulation on the network with 128 HCAs is 1,239,827. If we divide this number by the number of HCAs, we'll get about 9700 events per HCA during

the simulated time of approximately 10 microseconds. This is about 1000 events per simulated microsecond per HCA. If we do the same calculations for the other simulations, we'd get approximately the same results. For example on the network with 392 HCAs, the 45 sends per node simulation consisted of 11,785,702 events and simulated 30 microseconds. 11,785,702 divided by 392 is about 30,000, which is again about 1000 events per HCA per simulated microsecond.

If we look at the number of events during the simulation of the “idle” network (i.e. when HCAs don't send anything), we would discover that that this number is approximately equal to the number of events in the “busy” network multiplied by 0.56. In other words the number of events in the idle network is equal to 56% of the number of events in the busy network.

These simulations were run on an Intel Core2Duo T6600 @2.4GHz. The number of events per second on this hardware was between 200,000 and 240,000.

It is difficult to produce a precise equation for estimating the simulation time, but a rough estimate would be between

$$(t_s * P * 560 / E) \text{ and } (t_s * P * 1000 / E)$$

where t_s is the estimated simulated time in microseconds, P is the number of processes or nodes being simulated and E is the number of events in Omnet++ which the hardware, on which Omnet++ is running, is able to simulate per second. The real running time may be smaller depending on how active the simulated nodes are (how much traffic they produce.)

Here comes an example. Let's say we have a network with 8 HCAs and some switches. Usually we don't know precisely how long the simulated time is going to be, but there is always a rough estimate. This estimate can be 75 microseconds in our example. It is also hard to estimate the average activity of nodes, but we know that each of them will cause between 560 and 1000 events per simulated microsecond. So a simulation of 8 nodes for 75 microseconds will take maximum: 1000 events per simulated microsecond per HCA * 75 microseconds * 8 HCAs / 220,000 simulated events per seconds = 2.7 seconds (excluding the initialization phase which takes constant time depending on the simulated network size). The minimum time this example simulation will take is 56% of the maximum time, i.e. about 1.5 seconds. So this simulation will take between 1.5 and 2.7 seconds to run depending on how active the HCAs are. This figure may vary somewhat if the proportion between the number of switches and HCAs is very different from what I was using. The real simulation time for small networks may be smaller than that figure due to how CPU cache works (large networks take more memory, which means more frequent cache misses). Although not too accurate, this method may help us determine whether a simulation would take minutes or hours to run.

4.8.2 Summary

One thing to notice is that if we use nanosecond precision in the original integration setup the simulation is not exactly precise. The packets may arrive in-between nanosecond boundaries (as already mentioned, Omnet++ internally operates with picoseconds), and the order to send a new one will not be given by LogGOPSim before the next nanosecond. In large simulations like the one we did on M9 topology this error gets accumulated with the time and may become of considerable size (I observed up to 10% error). However, the pure Omnet++ simulation is almost as precise as a simulation can be, and so is the simulation run on the new setup. The way to make the original Omnet++ & LogGOPSim integration more precise is to decrease the base time unit.

Picosecond precision is the best we can get in Omnet++; however, the smaller time unit we use, the less efficient our simulation gets – the number of query messages increases proportionally with the decrease of time unit. The small simulations (dissemination among 8 nodes) which used to run almost instantly (0.6s) with nanosecond precision, took about 54 seconds to run with picosecond precision – such (in)efficiency is unacceptable for longer simulations.

As we can see the total running times excluding Generator times are different in the original integration setup and pure IB Model setup. In the pure Omnet++ setup it is about 15 seconds, while in IB Model & LogGOPSim integration it is about 34-15=19 seconds. One reason for this is that with Generator having to do more cycles (it's several million idle cycles alone!) there is more internal overhead for Omnet++ core. The other reason is that we in the IB Model & LogGOPSim integration there is one extra active process running in parallel - LogGOPSim, so the processor is more busy and the overall running time increases. One thing to mention about the pure IB Model simulation is that every traffic pattern/simulation has to be programmed separately. The integration of the IB Model & LogGOPSim is more general since the traffic pattern comes from the schedule file serving as input to LogGOPSim.

Optimizing the old setup gives at best 20% efficiency gain for dissemination among 648 nodes simulation on M9 topology (though this gain may differ for different simulations). At nanosecond precision this gain is less visible because of the fixed overhead while creating the "routing table". Manipulating precision affects the running time to a greater degree than manipulating the optimization level.

It is clearly visible that the new setup is more efficient in the sense of running time than the old. The running times are comparable for the old setup with nanosecond precision and the new setup, however nanosecond precision is not adequate for many simulations. The new setup is also more user friendly as it doesn't require manipulating with optimization and time granularity levels. Event log files for the new setup are much smaller than for the old setup because there are fewer messages (events) to log.

When it comes to scalability of the setup we can conclude that the simulation time increases proportionally with the increase of the number of simulated nodes and the increase in the simulated time.

It is hard to make a precise estimation of how long it would take to simulate something, however if one knows the number of nodes being simulated, the approximate final simulated time and the number of Omnet++ events one's hardware is capable of simulating per second, one can predict whether a simulation would take seconds/minutes/hours or days to run.

Chapter 5 Evaluation

In this chapter we perform several simulations using the integration, take a closer look at how the integration works, whether the simulations fit reality, and if not – we investigate what has to be done to improve them. The evolution of this research will be presented.

The chapter consists of three parts. First I explain the correspondence between the MPI traces, the .goal schedule, the simulation's course and correspondence to the real world. A few problems are uncovered during the first part. In the other two parts I evaluate the two possible solutions for these problems.

5.1 The Topology of the Cluster

All the tests in this chapter are run on a little computer cluster consisting of 8 HCAs and 6 switches. The topology has already been mentioned in section 4.7, and was then called H8_S6. This fat tree topology is shown below in Figure 5.1.

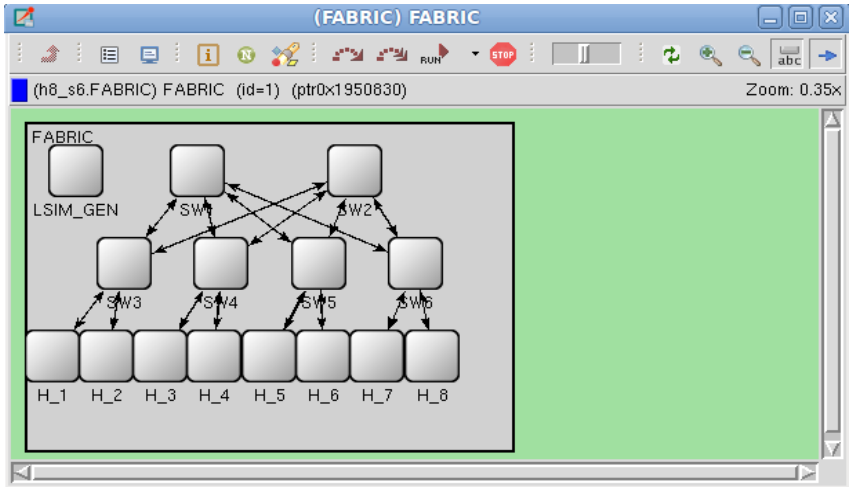

Figure 5.1. The fat tree topology used under the tests.

5.2 The First Simple Test: Trying to Understand What's Going on

5.2.1 The Test Program "Itest"

In section 4.7 we have already done some verification tests using a "synthetic" simulation schedule. In this section we are running a simple simulation based on the MPI traces. One reason for doing this is finding out whether our trace-driven simulation approach works and uncover any problems present. The other reason is illustrating the correspondence between the MPI traces, the .goal schedule produced from these traces and the course of the simulation based on this schedule.

A very simple MPI program has been used in this test. The source code (~40 lines) can be found in **Appendix B**. The program is supposed to run on two nodes. The nodes are playing "ping-pong" sending each other a buffer of 100,000 integers. The size of an integer is 4 bytes on the cluster on which ltest war run, so the size of the buffer is 400,000 bytes. During the running time

of the program each of the two nodes performs 10 sends and 10 receives in total, i.e. the total traffic is 8MB.

5.2.2 Interpreting the Trace Files

Two trace files, one from each node, were produced during the running of ltest. They can be found in **Appendix C**. Let's look a bit closer on the contents of the first trace file.

We ignore the lines starting with '#'-symbol, and start reading from the line starting with "MPI_Init". The first three lines (see Figure 5.2.2.1) represent the calls to the three MPI functions. These three function calls are standard for every MPI application. The long number in the end of each of these lines (starting with digits 130...) is the return time from the corresponding function. All the times in traces collected by liballprof are the numbers of **microseconds** from the start of epoch. All the numbers that do not represent call or return times are parameters to the corresponding functions. Call times, return times and the function arguments are separated by colons. Some arguments may consist of several values separated by commas.

```
MPI_Init:-:140735615211452:140735615211440:1302767374553965
MPI_Comm_rank:1302767374553992:6575584,0,2:140735615611496:1302767374554004
MPI_Comm_size:1302767374554016:6575584,0,2:140735615611492:1302767374554026
```

Figure 5.2.2.1 The first three lines of the trace

The subsequent 20 lines represent the 10 send and 10 receive calls. The first 2 of these lines are in Figure 5.2.2.2. Notice, that the first such call for the first node is a send, while the first such call for the second node is a receive. The first and last numbers in these lines are function enter and return times.

```
MPI_Send:1302767374554038:140735615211488:100000:1,4,4:1:0:6575584,0,2:1302767374555673
MPI_Recv:1302767374555688:140735615211488:100000:1,4,4:1:0:6575584,0,2:140735615211456:1302767374555978
```

Figure 5.2.2.2 An MPI_Send and MPI_Recv lines from the trace

The last line in the trace is "MPI_Finalize:1302767374567772:-". It represents the call to MPI_Finalize function and contains the function's return time.

Unfortunately the system clocks on the two nodes aren't necessarily synchronized. This means that to calculate the time it takes to send the message over we can't just take a difference between entrance time into a send function on one node and return time of the corresponding receive function on the other node. However, in our simple MPI program the nodes are playing ping-pong with a 400,000 bytes long message. So the difference between entrance time into a send and return time of the immediately following receive is approximately equal to the time it takes to send the message back and forth (usually there is a 5-15 microseconds gap between MPI_Recv and MPI_Send – this time is also included in the "round-trip" time, which is not perfectly correct, though the error is small compared to the round-trip time). Half of this round-trip time is the approximate time it takes to send the message once. Let's take the last (10_{th}) send-receive sequence of the first node as an example (on the bottom of the trace file). The send was called at time 1,302,767,374,560,454. The following receive returned at 1,302,767,374,561,003. The difference between these two times is 549 microseconds. This means that it takes about 275 microseconds for a node to send the message to the other node. If we make the same calculations on the other send-receive or receive-send sequences the results will be approximately the same. The exception is the very first such sequence, which takes much longer time. The reason for this exception will be explained and discussed later, in the end of this section. All ten message travel

time calculations can be found in Table 5.2.2.

Send-recv seq #	Send-recv seq start	Send-recv seq end	Time difference	Half of time diff
1	1302767374554038	1302767374555978	1940	970
2	1302767374555987	1302767374556538	551	275.5
3	1302767374556546	1302767374557095	549	274.5
4	1302767374557104	1302767374557654	550	275
5	1302767374557663	1302767374558211	548	274
6	1302767374558220	1302767374558768	548	274
7	1302767374558777	1302767374559325	548	274
8	1302767374559334	1302767374559881	547	273.5
9	1302767374559890	1302767374560445	555	277.5
10	1302767374560454	1302767374561003	549	274.5

Table 5.2.2 MPI_Send call times, subsequent MPI_Recv return times, difference between them ("round-trip time", half of round-trip times) in microseconds

By using the trace we can also find out the running time of the program. It is the difference between return time from MPI_Init and return time from MPI_Finalize. In our case, using the first trace, the running time is 13,807 microseconds.

What about that first call, the one that took unusually long time? The MPI_Send function pins the buffer provided by the user, so that the buffer is not swapped to disk by the operating system (the other data transfer MPI functions also pin the user provided buffers). The pinning happens when the buffer is provided for the first time. The pinning doesn't happen when the same buffer is provided during subsequent calls to MPI_Send. In our test program we're reusing the same buffer in all MPI_Send (and MPI_Recv) calls. Pinning the memory is the reason why the first call to MPI_Send is much slower than the subsequent ones. A small experiment was performed: the code was slightly changed, so that the fifth send used a new buffer. During this experiment the fifth send also became slow, which proves that the extra delay is caused by using the previously unused buffer.

The manual reading of the MPI traces has uncovered the problem with memory pinning which causes the transfers of the previously unused buffers to take longer time. This issue was not taken into account in the design of the integration and has to be dealt with.

5.2.3 Looking at the .goal Schedule

The complete .goal schedule file produced by Schedgen1.1 can be found in **Appendix D** and a short snippet is shown in Figure 5.2.3. Notice, that the time units used in local operations ("calc") are **picoseconds**, because Schedgen1.1 was run with an argument which makes Schedgen1.1 convert microseconds from the trace into picoseconds in the resulting .goal schedule. Let's take a closer look at the schedule for the first node and try to find links to the trace file. First in the schedule there is a send operation followed by a local operation (local operation represents processing). Then on the third line we see that the send requires the local operation to complete before the send can start, i.e. local operation is simulated first, then we simulate the send. The duration of the local operation is 73 microseconds (73 million picoseconds). If we take the call times of MPI_Init and the first call to MPI_Send from the trace file, we will discover that the difference between them is exactly 73 microseconds... The amount of data sent by all the send operations (and received by all the receive operations) is 400,000 bytes, which is equal to the size of the message – it can be seen both in the test application source

code and in the trace file.

```
rank 0 {
l1: send 400000b to 1 tag 0
l2: calc 73000000
l1 requires l2
l3: recv 400000b from 1 tag 0
l4: calc 15000000
l3 requires l4
l4 requires l1
l5: send 400000b to 1 tag 0
l6: calc 9000000
l5 requires l6
l6 requires l3
< . . >
l41: calc 6769000000
l41 requires l39
}
```

Figure 5.2.3 A short snippet of the ltest trace file.

The destinations and tags of the send and receive operations in the schedule also clearly originate from the trace. If we look at the dependencies ("requires"-lines) in the schedule, we see, that they ensure the sequential execution of sends and receives with relatively short local operations in-between. The first such short local operation is marked "*l4*" and lasts for 15 microseconds. Not surprisingly this is exactly the period of time between the return of the first send and call to the subsequent receive. In the end of the schedule for the first node we can find a local operation with a duration of 6769 microseconds. This is the time difference between the return time of the final receive operation and the return time of MPI_Finalize.

5.2.4 Simulating "ltest"

In this section the ltest simulation conditions and results are presented. The small size of this simulation makes it possible to look into all the details, compare the simulation's course to the MPI traces and find any inconsistencies.

Two simple log files were written during simulation. They can be found in **Appendix E**.

- The first log file was written in the Generator module – each line in this log file represents a message insertion into the network, and contains message unique ID (handle), simulated time in Omnet++ (simTime() in seconds), simulated time at which the inserted message is supposed to start traveling through the network (currtime_l in picoseconds), message source and destination. A line from this log file may look like this:

  ```
  insert handle: 19 simTime(): 0.00537324935 currtime_l: 5437249350 src_l: 1 dest_l: 0
  ```

- The second log file was written in the sink module, where the arrived messages are consumed. Each line represents an arrived message and contains the message ID and simulated arrival time in seconds. A line from this log file may look like this:

  ```
  sink: message 19 has arrived at 0.005654847
  ```

The LogGOPSim was run with parameters L=0, g=0, G=0, o=55,000,000 (the reason for using this value for 'o' will be explained below). If we look for example at the first lines of the log files, we'll find out that the message 0 has started traveling through the network at time 73 microseconds from simulation start, and arrived at destination at time ~291 microseconds from

simulation start. This means that it traveled for about 218 microseconds. When we looked at the trace, we found out that the typical traveling time for a message was about 275 microseconds. The reason for this inconsistence is that we're simulating Infiniband link layer in detail, application layer is represented by the local operation delays in LogGOPSim, while the network and transport layers aren't simulated at all. This is the reason for which we used o parameter equal to 55 million picoseconds (55 microseconds) – to compensate this inconsistence. Later we'll discuss better ways of doing it.

As already mentioned message 0 has arrived at destination 291,286,850 picoseconds (~291 microseconds) after simulation start. Message 1 which is supposed to start traveling right after the arrival of message 0, truly enough is inserted at the same time as message 0 arrives. However it doesn't start traveling through the network until 361,286,850 picoseconds (~361 microseconds). If we take a look into the goal schedule for node 1 (which is the source of message 1) we'll see that there is a local operation with duration of 15 microseconds between the reception of message 0 and sending of message 1. In addition we have a processing overhead (o-parameter) set to 55 microseconds. These two delays together sum up to 70 microseconds, which is exactly the difference between 291 and 361 microseconds.

On completion LogGOPSim has printed the completion times of the two nodes to be 12,478,847,000 picoseconds (~12.48 milliseconds) and 12,454,847,000 picoseconds (~12.45 milliseconds) respectively. In the arrivals log file we see that the last, 20th, message has arrived at the first node at 5,654,847,000 picoseconds (~5.66 milliseconds) after simulation start. In the end of the schedules for each of the nodes we can find local operations with duration around 6.8 milliseconds. Adding this delay to the last packet arrival time will get us to the time printed out by LogGOPSim. When looking at the trace file we found out that ltest actually ran for about 13.8 milliseconds. There is an inconsistence between this time and time printed out by LogGOPSim of about 1.3 milliseconds. This can be explained by the memory pinning delays. In our case we had one such delay at each node. Each delay was about 600 microseconds, so the two of them explain most of the inconsistence.

5.2.5 Problems Discovered During the First Test

There were two problems which became visible when simulating the simple MPI program with the integration.

The first problem concerns pinning of memory every time a new user buffer is provided to a data transfer function in MPI (like MPI_Send). We knew exactly what was going on in the simple test program – there was just one pinning delay with known duration on each node, so it could be easily explained. However there is no easy way to get this information for a **general** MPI application; we can't know how buffers are used (reading source code of every MPI application simulated is an ineffective solution). I've tried two solutions (the first simple solution didn't work, so the second one had to be applied).

The first (and simplest) solution involves compiling Open MPI Library using "--without-memory-manager" option [31], which is supposed to exclude the memory pinning. Notice, that this solution actually means altering the test application so that the simulation fits, instead of altering the simulation. With this change of the MPI Library the duration of the first send in ltest became about 1600 microseconds (vs. 970 microseconds for the build with memory manager enabled), the duration of a regular send became about 650 microseconds (vs. 275 microseconds

for the build with memory manager enabled). The reason for this is connection establishment.

The second solution involves a few small changes to the Schedgen1.1 source code. What has been done is implementing an algorithm for checking when a previously unused buffer is provided to a data transfer MPI function, and then adding an extra local operation with a duration to compensate for pinning of the buffer. The local operation is added in such a way, that it executes right before the data transfer operation. The C++ implementation of the algorithm for checking which buffers have previously been used is supplied in the **Appendix F**.

How pinning delay changes depending on the size of buffer to be pinned has been determined in the experimental way, by taking the difference between the durations of MPI_Send being called with a previously unused buffer, and the already used buffer. The experimental data is presented in Table 5.2.5.1. This data is hardware dependent and may differ on different clusters.

Using linear regression on the data presented in Table 5.2.5.1 resulted in the following function: f(x) = 165.894 + (0.00301695 * x). The implementation of linear regression analysis can be found at [32]. Naturally a zero size buffer should take no time to pin, however we see that it takes 165 microseconds according to the deduced function. This is caused by the linear regression error. In general case we can assume that increasing the buffer size by 1 byte leads to about 3 nanoseconds pinning delay increase.

Datasize in bytes	Duration of the first send	Duration of the subsequent send	Difference
4	32	14	18
16000	279	35	244
32000	343	46	297
48000	395	56	339
64000	447	64	383
80000	517	73	444
160000	792	126	666
240000	1092	176	916
320000	1316	224	1092
400000	1635	272	1363

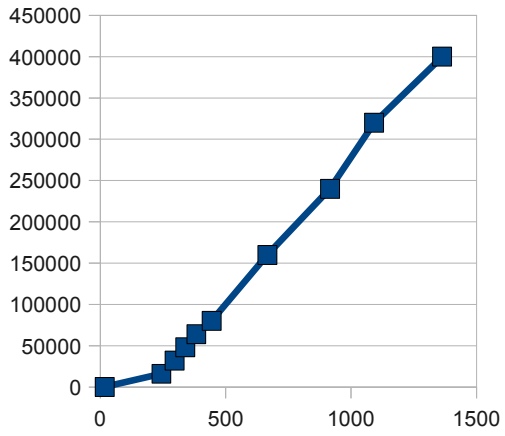

Table 5.2.5.1 MPI_Send durations for used/unused buffers of different sizes

The second problem concerns the time spent on layers 3 to 5. It is the most important of the two problems.

The integration consists of a detailed link layer simulation in Omnet++ and a simple application layer simulation in LogGOPSim (local operations in LogGOPSim). This is illustrated on Figure 5.2.5. The time spent on the layers which are not simulated still needs to be compensated for. In the trace, which is collected between the MPI and the Application layers, this time is included between the call and return time of an MPI data transfer function (MPI_Send for instance). When simulating ltest, we were using the fixed overhead per message parameter ('o' of the LogGOPS model), which solved the problem for that particular test, however a general solution is needed.

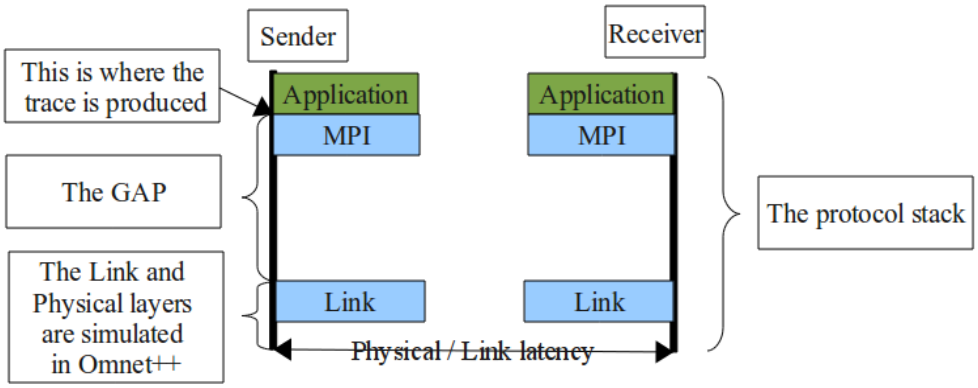

Figure 5.2.5: which parts of the protocol stack which are not covered by the simulation

5.3 Using the 'o' and 'O' Parameters in the Simulation

As mentioned in section 5.2.5 the simulation setup requires a compensation for the time spent by a data message in layers 3 to 5. This compensation varies with the message size, so if we want to utilize the LogGOPSim's capability of using the LogGOPS model to solve this problem, it is more appropriate to use both the processing overhead per byte and per message parameters of LogGOPSim ('O' and 'o' respectively). All that remains is determining these parameters.

As already stated in section 2.7.5, two different protocols are used for sending small and large messages. In the experiment described below we prove, that the data messages use different time to travel down the protocol stack depending on the protocol used. This means that we should use two different sets of 'o' and 'O' parameters – one set for small messages, and another set for large messages. In LogGOPSim we have just one set of parameters used no matter which protocol is applied.

To solve this minor problem the LogGOPSim source code has been slightly changed, to utilize two sets of parameters. The processing and networking resources are charged only in two places in LogGOPSim code: in the event handlers for the send and Message-on-flight events (Section 3.1). The change to source code involves checking message size before charging resources and then charging them based on the appropriate parameter.

To determine the processing overheads required for compensation multiple tests have been performed running the same simple MPI program (ltest) with different message sizes. Then the simulation has been run based on the traces collected. In Tables 5.3.1 and 5.3.2 there is a summary of real message travel times and simulated message travel times for different message sizes (simulated without compensation for layers 3-5).

Datasize in bytes	Real msg travel time [µs]	Simulated msg travel time [µs]	Difference
4	23	0.1115	22.8885
16000	40	8.8	31.2
32000	50	17.5	32.5
48000	60	26	34
64000	69	35	34

Table 5.3.1 Real vs simulated message travel times for small message sizes

Datasize in bytes	Real msg travel time [µs]	Simulated msg travel time [µs]	Difference
80000	81	44	37
160000	129	87	42
240000	180	131	49
320000	226	174	52
400000	276	218	58

Table 5.3.2 Real vs simulated message travel times for large message sizes

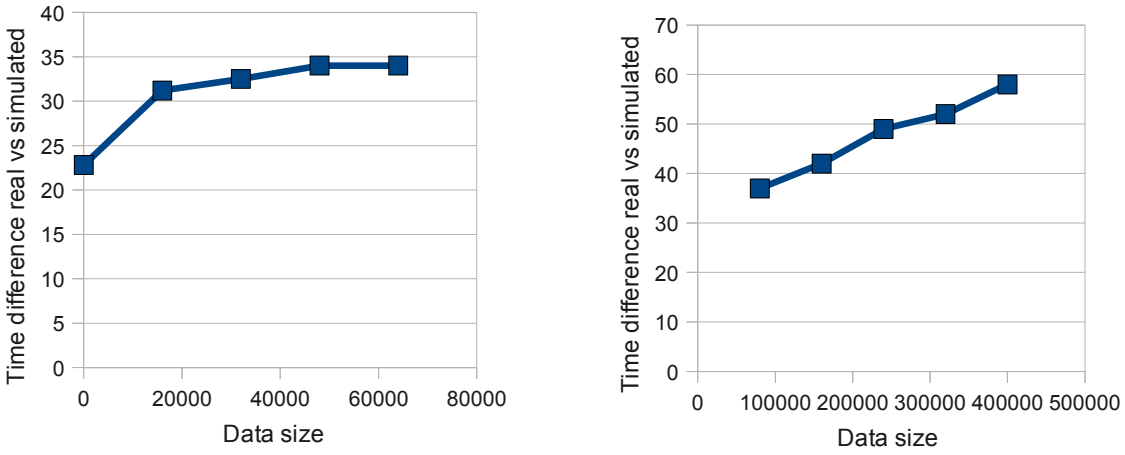

Table 5.3.3 Charts of differences between real and simulated travel times for small and large messages of different sizes

Using linear regression on the data presented in the two charts in Table 5.3.3 it is possible to find arguments for the functions describing the dependency between the message size and the required compensation. Notice that these arguments are valid only for simulations based on traces collected on the cluster at Simula Research Laboratory – things will be different on other hardware. The function is linear (b+ax) to reflect the influence of 'O' and 'o' parameters in LogGOPS model. The arguments 'a' and 'b' in our case will be the same as respectively 'O' and 'o' of the LogGOPS model. The function for small data sizes is f(x) = 25.9798 + (0.000155003 * x). For large data sizes it is f(x) = 32 + (0.000065 * x). The function for small data sizes in verbal form would be something like "a zero size message travels for about 26 microseconds, and if the size is increased by one byte the travel time increases by about 155 picoseconds".

Simulating ltest with small messages, 'O'=155 and 'o'=25.979.800 gave the following results:

Datasize in bytes	Real total run time [µs]	Simulated total run time [µs]	Difference
4	6585	6763	-178
16000	6963	6746	217
32000	7463	7236	227
48000	7530	7251	279
64000	8857	8589	268

Table 5.3.4 Results of simulations with small messages

Simulating ltest with large messages, 'O'=65 and 'o'= 32.000.000 gave the following results.

Datasize in bytes	Real total run time [µs]	Simulated total run time [µs]	Difference
80000	9148	8813	335
160000	9370	8784	586
240000	10557	9703	854
320000	12059	11059	1000
400000	13807	12539	1268

Table 5.3.5 Results of simulations with large messages

The error is caused by inaccuracy of linear regression to a lesser extent, and memory pinning delay to a larger extent. The larger the buffer to be sent, the more memory pages are to be pinned and the longer time it takes. This explains that the error is increasing towards the larger message sizes.

5.3.1 Taking a Closer Look at the Collective Operations

At this point we have obtained two sets of processing overheads that can be used to compensate for the message traveling time down through layers 3-5. These overheads were deduced from point-to-point operations. However, MPI supports not only point-to-point, but also collective operations. So we need to investigate whether the overheads will work for simulating the collective operations.

Another little MPI application has been used here (the source code is in **Appendix G**). The application is run on 4 cluster nodes. The trace for the first cluster node can be found in **Appendix H**. At first the nodes with even ranks do a single ping-pong "exchange" with the odd nodes using point to point operations (so that the buffers are pinned and connections are established before doing the collective operations). Then the two collective functions MPI_Allgather and MPI_Allreduce are called 10 times each, with gradually increasing buffer sizes. (MPI_Allgather gathers data from all processes and distributes it to all processes. MPI_Allreduce performs some operation on data from all processes and distributes the results to all processes.) These calls are alternated with the MPI_Barrier calls.

As we can see from the trace in **Appendix H**, the application ran for 32,148 microseconds on the first cluster node, almost a quarter of which (7753 microseconds) is the time between the last MPI_Barrier and the MPI_Finalize. The final simulated time (when using the deduced processing overheads as LogGOPSim parameters) was only about half of that: 16,408,145,370 picoseconds or 16,408 microseconds. Considered that 7753 microseconds out of this time is finalizing this result is very wrong. This proves that the previous approach, i.e. trying to use some processing overheads deduced from point to point operations, is not perfectly correct for general MPI applications.

The Schedgen (.goal schedule generator and trace parser) supports five MPI collective operations: MPI_Allreduce, MPI_Allgather, MPI_Bcast, MPI_Reduce and MPI_Alltoall. When trying to understand why the overheads do not work, the first idea was trying to understand the difference between the point to point and collective operations by looking at collective function durations for different buffer sizes and number of nodes. Then this information could be used to further calibrate the simulation setup to fit reality. Let's take a closer look on the durations of function calls from the trace in **Appendix H**. These are presented in Table 5.3.1.1.

As we can see there is an overall pattern in the function durations for different data sizes. The durations generally increase when the data sizes increase. This is especially obvious for the MPI_Allgather durations. However there are some durations which clearly don't fit the pattern. If we go back to the source code in **Appendix G** we'd find out that the collective operation calls are preceded by a ping-pong exchange in which the same buffers were used. So these buffers have already been pinned in memory (which explains the relatively long durations of MPI_Send calls). This means that we shouldn't have any memory pinning delays when reusing the same buffers in the collective functions. The duration of the first Allgather call can be explained by connection establishment (not all the connections were established during the preceding ping-

pong exchange). However, there is no good explanation for the deviating durations of MPI_Allreduce for data sizes 11,200 and 14,400 bytes. Notice, that the duration of

Allgather

Data size in bytes	Function duration [microsec]
1200	3527
2400	53
3600	59
4800	75
6000	71
7200	71
8400	81
9600	85
10800	85
12000	98

Allreduce

Data size in bytes	Function duration [microsec]
1600	65
3200	54
4800	67
6400	72
8000	100
9600	108
11200	13735
12800	120
14400	522
16000	146

Table 5.3.1.1 The durations of consecutive MPI_Allgather and MPI_Allreduce calls with different buffer sizes

MPI_Allreduce is equal to 13,735 microseconds, which is quite considerable compared to the total application running time which was 32,148 microseconds. The length of these 3 calls (one call to MPI_Allgather and two calls to MPI_Allreduce) also explains the large deviation between the running time simulated using the processing overheads of LogGOPSim and the real running time.

So the idea with processing overheads may be not that wrong anyway. If these unusually wrong collective calls had normal duration fitting the overall pattern, the application would have taken about 14,600 microseconds to run, or about 6900 microseconds if we exclude the finalize duration. In the simulation it took 8653 microseconds excluding the finalize duration, in other words we have about 20% error if we don't take the unusually long calls into account.
Whether 20% error is considerable or not depends on the amount of collective operations in the particular simulation and what the person performing the simulation wants to achieve (if only a small fraction of operations in the simulation are collectives, then the error in the total simulated running time would not be large). However, I have another idea of how this problem with message traveling times through layers 3-5 may be solved. This idea is expected to solve the problem with both pinning times and the unpredictably long durations for some collective operation calls too. This other approach is explained in section 5.4.

5.4 Using MPI Function Durations as Local Calculations

When taking a closer look at some of the MPI collective functions it became obvious that it is very hard to predict the durations of the collective functions, as we did with the point to point operations. A new approach was invented.

Let's take a fresh look on the problem at hand. On Figure 5.2.5 we can see an incomplete protocol stack. The application layer of this stack represents the time spent outside the MPI functions, i.e. the time between return from one function and call to the subsequent function. If we have for example an MPI_Send which returns at time 100, and then an MPI_Allgather being called at time 115, then we'll have a local calculation operation in the .goal schedule with duration 15, and some dependencies ensuring that this calculation happens between the send representing MPI_Send and the first operation in the sequence representing MPI_Allgather. This is how the application layer is “simulated”. Whenever a send operation is simulated, an insert

into the network takes place. The network is simulated by the IB Model, however, the IB Model simulates only the link and physical layers. This means that we have a gap between the top of the MPI layer, which is just below the application layer, and the top of the link layer. This gap is not simulated (though we've tried to fill it using some processing overheads before with limited success).

Suppose an MPI_Send function is called to send 1 byte of data. It probably takes some time between MPI_Send is called and the data actually starts traveling through the physical wire. During this time the message "travels" all the way down the stack. It is obvious that the whole way down reflected in the duration of MPI_Send is larger than just the time it takes to travel through the link layer. This time difference needs to be simulated. Previously we've been trying to find some system or regularity in this. It didn't quite work out for the collective operations – truly enough, the different MPI functions are doing different things, so even if the common system exists, finding it would be hard.

However, it is not unreasonable to assume that **the "gap" duration is equal to the duration of an MPI function called minus the time it takes all the data to travel through the link layer**. This time can be inserted as a local calculation operation in the .goal schedule to fill the gap. Implementing it is really simple since we already have the pinning delay insertions, all that remains is using the appropriate time for these delays. Notice, that this solution completely eliminates the pinning delays and connection establishment delays problem – all these delays are included in the duration of the MPI function, and thus will be included in the duration of the local operation. All we need is the link layer times.

Fortunately we have a detailed link layer simulation in Omnet++. This simulation is calibrated to fit the little research cluster at Simula Research Laboratory, the one on which the traces are collected. Several tests have been run, during which messages of varying sizes were sent. The goal was to find several different times. There are four "checkpoint" times which are of interest to us:

1. the flit is "generated" by the gen simple module (on behalf of the Generator)
2. the flit leaves the obuf simple module
3. the flit arrives at the destination's ibuf simple module
4. the flit is consumed at the destination's sink

We are primarily interested in the times between checkpoints 1 and 2, and checkpoints 3 and 4, which are the time it takes a flit to travel down through the link layer (during sending), and the time it takes a flit to travel up through the link layer (during receiving). The method for finding these was producing a log file in each of the simple modules we're interested in, and then doing some basic maths (mostly subtraction). The linear regression was not required here as all the delays in the IB Model in Omnet++ are either constant or follow a clear pattern.

When it comes to making a simple analytical model to calculate the link layer delays, the following was found:

- it takes 0 picoseconds for the first or only flit of a message to travel down through the link layer (send)
- it takes 32.000 picoseconds for every subsequent flit to be sent down through the link layer
- it takes 33.600 picoseconds to receive a 64 byte large first flit of the message, and this time increases by 1.6 nanoseconds for every subsequent flit of the message
- if the message consists of just one flit, and it is shorter than 64 bytes, than the receiving

time time is a proportional fraction of 33.600 picoseconds (for example 16.800 picoseconds for a 32 bytes large flit).

So the sending time is 32.000 * (number of flits – 1). The receiving time for a multi-flit message is the sum of several members of an arithmetic progression with the first member equal to 33.600 and the increase equal to 1600. The number of members is the number of flits.

Naturally the analytical model above gives just an approximation, the actual IB Model in Omnet++ is more complex than that. For example we don't take account of network congestion in our simple analytical model. However, usually the link layer times are relatively small compared to the durations of the MPI_Functions, so even being not completely accurate sometimes, these link layer times introduce little error to the simulation, and the overall simulated traffic pattern should resemble the real one.

Another thing worth mentioning is that Schedgen converts the collective operations from traces into series of sends and receives. The number of sends and the number of receives in a series are known, the sizes of messages being sent are given too. So the delay (duration of the local operation) for the collective operation in the .goal schedule would be:

*return time - call time - (number of sends * sending delay) - (number of receives * receiving delay)*

where the sending and receiving delays are calculated based on the analytical model described above.

5.4.1 Does the New Approach Work?

The first test which was run after implementing the approach described above was the NASPB MG class S, which is the shortest of all NASPB tests (Section 5.5). The real running time for the first process was 62,737 microseconds (or ~63 milliseconds). The sum of the durations of all the local calculations for rank 0 is 56,306,155,910 picoseconds (or ~56 milliseconds), which is naturally a bit smaller than the real running time – the difference is supposed to be spent in the link layer, so that the total simulated running time is about the same as the real one. However the final simulated running time for the first process turned out to be 74,631,205,354 picoseconds (or ~75 milliseconds), which is about 20% larger than what it was supposed to be. If we run this simulation using LogGOPSim alone, we'll get the final simulated time of 74,531,154,009 picoseconds, which is only about 100 microseconds smaller than the simulated time achieved when using the “integration”.

Obviously something is not completely right here. The trace file for the NASPB MG class S is about 1500 lines long, and the .goal schedule consists of 4 ranks, about 8500 lines each. This is a bit too large for manual reading and trying to understand what is going on. Let's use another little MPI application which also leads to “wrong” results. The source code, traces and the .goal schedule can be found in **Appendix I**. The schedule presented there was produced using the calculation delays after subtracting the link layer delays. There is also a schedule, which is not presented here, where the link layer delays have not been subtracted. There are four different times:

1) real total running time: 6,721,000,000 picoseconds
2) sum of all calculation delays with link layer delays subtracted: 6,720,868,800 picoseconds
3) simulated total time with the “integration”: 6,731,069,500 picoseconds
4) simulated total time with LogGOPSim alone: 6,730,955,900 picoseconds

We can see that the simulated time is always larger than the real running time, even when simulating with LogGOPSim alone. The simulated time with LogGOPSim alone is not equal to the sum of all delays as one would expect. The error is about 10 microseconds.

	Rank 0	Rank 1	
Start	0	0	Start
+Calc	79000000	74000000	+Calc
+Calc	33968500	34968500	+Calc
=receive at	112968500	108968500	=receive at
+Calc	13000000	13000000	+Calc
+Calc	19000000	19000000	+Calc
=send at	144968500	140968500	=send at
+Calc	13000000	13000000	+Calc
=msg at	157968500	153968500	=msg at
+Calc	17000000	32000000	+Calc
+Calc	13000000	12000000	+Calc
=send at	187968500	197968500	=send at
=receive at	187968500	197968500	=receive at
!=msg at	198082100	197968500	=msg at
+Calc	42987400	77987400	+Calc
+Calc	6490000000	6472000000	+Calc
=total	6731069500	6747955900	=total

	Rank 0	Rank 1
init start	0	0
irecv call	79	74
irecv ret	34	35
send call	13	13
send ret	19	19
wait call	13	13
wait ret	17	32
allreduce call	13	12
allreduce ret	43	78
fin	6490	6472
total	6721	6748

Table 5.4.1 The "log" of the simulation run on the integration and a trace summary

In **Appendix J** the detailed output from LogGOPSim is presented. It's slightly more readable summary is presented in Table 5.4.1.

On the right hand side of the table we can see a very brief summary of the two trace files. All the numbers here are the numbers of microseconds. It says that MPI_Irecv is called 79 microseconds after the start of the first process. MPI_Irecv returns 34 microseconds after it is called. MPI_Send is called 13 microseconds after MPI_Irecv returns, etc. Both processes run relatively synchronized in the start, i.e. the same things happen approximately at the same time. The first relatively large "dis-synchronization" happens during the call to MPI_Wait, which lasts longer for rank 1 (32 microseconds for rank 1 versus 17 microseconds for rank 0). Notice, that the actual message reception happens some time during the MPI_Wait call.

On the left hand side of the table there is a summary of what LogGOPSim prints out during the simulation (when using the "-v" option). All numbers here are the numbers of picoseconds. First there is a delay of 79 million picoseconds for rank 0, and 74 million for rank 1, which are clearly visible in the "right hand side table". The next calculation's time is 31,500 picoseconds smaller than the duration of the corresponding MPI_Irecv call – this is the Link Layer delay for sending 1 flit which was subtracted in the schedule generator (the time for receiving a single flit message is proportional with the flit size, in our case it is 40 bytes of payload plus 20 bytes header, so the reception time of the link layer is 33,600 * 60 / 64 = 31,500 picoseconds, where 33,600 is the time it takes to receive a 64 bytes large flit). The receive is posted into the receive queue, i.e. this is not the actual reception of the message – the message hasn't even been sent yet. Then there is a 13 million picoseconds delay which represents the gap between the return from MPI_Irecv and call to MPI_Send. The next delay represents the MPI_Send duration after link layer sending delay subtraction. Then there is another inter-function gap, and the messages sent by MPI_Send are finally received. Notice, that by this time we are already "inside" MPI_Wait – the delay preceding the MPI_Wait call has passed. According to the log written in Omnet++, the first message (the one sent by rank 1) is received at time 141,101,000 picoseconds, i.e. 132,500

picoseconds after it is sent (101,000 link delay plus 31,500 Link Layer reception delay). The traveling time for the message sent by rank 0 is the same. After the actual reception of the message there are two delays: the first one represents the duration of MPI_Wait function call, and the other represents the gap between MPI_Wait and MPI_Allreduce. In our case there are just two processes participating, so the MPI_Allreduce sequence is just one send, one receive and a delay to represent the function's duration. Because rank 1 has spent too much time in MPI_Wait, it enters MPI_Allreduce 10 microseconds later, than rank 0. This means that the message, which rank 1 is supposed to send, is sent 10 microseconds later, and so it arrives at rank 0 10 microseconds later too. This is the reason for which we have a 10 microsecond error in the total simulated running time for rank 0. See Figure 5.4.1 for a comprehensible comparison of how things were in the real world and how they were simulated.

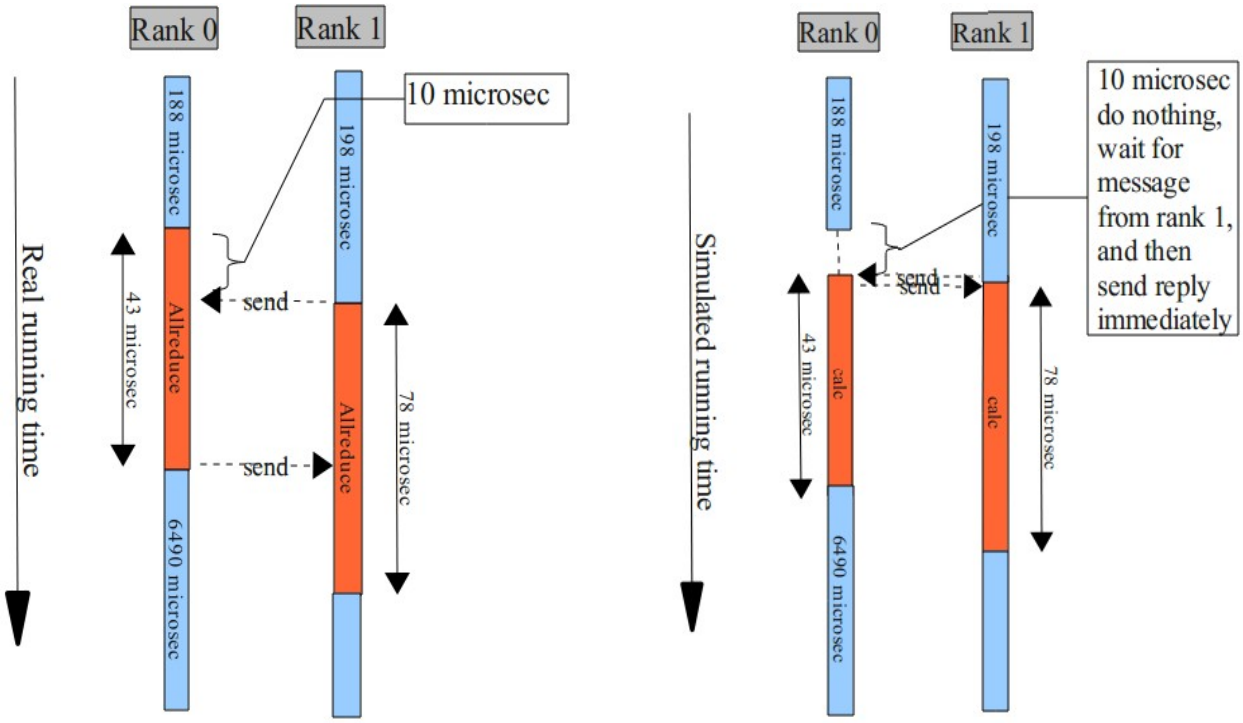

Figure 5.4.1 A comparison of the real world situation and how it was incorrectly simulated

Another thing worth noticing is that there is a 113,600 picoseconds difference between the simulated running times for rank 0 when using LogGOPSim alone and the “integration”. 113,600 picoseconds is the exact link layer traveling time for that “special” 24 bytes message (4 bytes payload, 20 bytes header) which arrived out of sync; in other words which has arrived not during a local operation delay. The number 113,600 is equal to the link delay of 101,000 plus the reception time of (24/64 * 33,600).

The conclusion here is that the error is caused by dis-synchronization between the ranks, which happens when the ranks enter an operation at different times, which causes one or several ranks to wait for the other(s). If all the communication happened when the function durations for different ranks overlap, this problem would be avoided. However implementation of this would require some quite fundamental changes in the functioning of the schedule generator. Currently the schedgen parses the trace files one by one and writes the schedules sequentially. To avoid the dis-synchronization described above, we would need to read all trace files in parallel, keep account of all the delays for each rank, and when handling a collective operation add a delay

before the communication sequence in such a way that these sequences start at the same time for all ranks. This way, if we use the example from above, we would know that MPI_Allreduce for rank 0 is entered 79+34+13+19+13+17+13 = 188 microseconds after the start, while MPI_Allreduce for rank 1 is entered 74+35+13+19+13+32+12 = 198 microseconds after the start (see the right hand side of Table 5.5.1). This way we could split the delay representing MPI_Allreduce duration for rank 0 into two delays, the first of which would be 10 microseconds and compensate for the late MPI_Allreduce entering of rank 1, while the second part of the delay would be 33 microseconds, so that the total duration of 43 microseconds stays constant.

It is worth noticing, that though the error was relatively small for the presented example, the example was tiny as well. For longer applications the dis-synchronization situations may happen multiple times, which increases the error. Things may get even worse when the application is run on many nodes instead of just 2 as in our example. It happens sometimes that MPI functions take thousands of microseconds to complete (we've already seen an example of MPI_Allreduce that took over 13,000 microseconds) – the dis-synchronization might get quite considerable.

5.4.2 Running More Tests

In this section results from several sets of tests will be presented. These tests were run using the new approach, when the function durations were used in the .goal schedule as local calculations. The first set of tests uses the already mentioned "ltest" application (**Appendix B**), in which the two nodes are playing ping-pong with data messages. This test was run on 3 different topologies using 10 different message sizes – 30 tests in total. One of the topologies is presented on Figure 5.4.2.

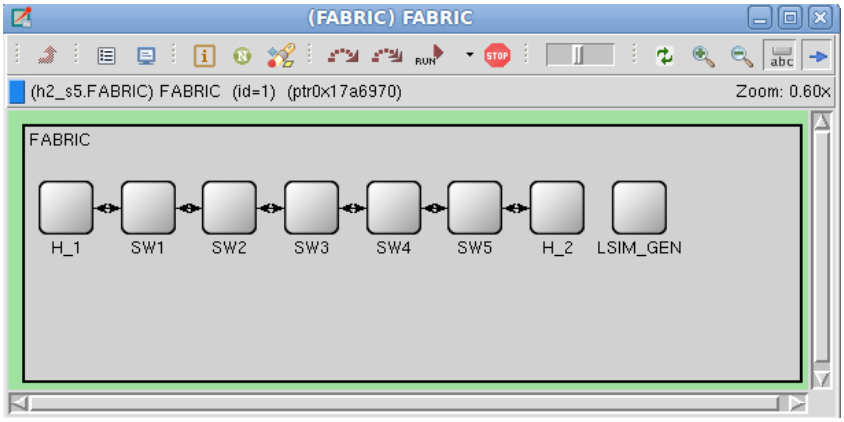

Figure 5.4.2 A network topology with 2 hosts and 5 switches connected in a chain.

The other two topologies are very much alike the one shown above, except that instead of 5 switches they contain 4 and 3 switches. The message sizes were 4, 16.000, 32.000, 48.000, 64.000, 80.000, 160.000, 240.000, 320.000 and 400.000 bytes. The results from the 30 tests performed are presented in Table 5.4.2. The difference is calculated on the basis of the real and simulated total running times excluding the finalize phase. These differences may seem relatively small, however considering the number of operations in these tests (10 sends + 10 receives = 20 operations) the error is considerable enough. Usually we're talking about one or two dis-synchronized operations taking place in each test.

3 switches

Datasize in bytes	Real total run time [µs]	Simulated total run time [ps]	Finalize time [µs]	Difference [%]
4	6454	6504181600	5861	8.5
16000	7886	7924865750	6641	3.1
32000	7342	7389103250	5861	3.2
48000	7897	7969124350	6161	4.2
64000	7961	8040699350	5991	4.0
80000	8329	8426597000	6117	4.4
160000	9313	9483130600	5850	4.9
240000	10533	10774959500	5845	5.2
320000	13025	13337755600	7093	5.3
400000	13211	13600361850	6006	5.4

4 switches

Datasize in bytes	Real total run time [µs]	Simulated total run time [ps]	Finalize time [µs]	Difference [%]
4	7027	7079275600	6455	9.1
16000	7345	7390149999	6082	3.6
32000	7909	7964499998	6421	3.7
48000	7757	7825190798	6008	3.9
64000	7978	8058878000	6015	4.1
80000	8537	8623240599	6309	3.9
160000	9590	9733031000	6147	4.2
240000	11313	11516808199	6599	4.3
320000	12851	13113595599	6950	4.5
400000	13544	13865371998	6302	4.4

5 switches

Datasize in bytes	Real total run time [µs]	Simulated total run time [ps]	Finalize time [µs]	Difference [%]
4	7108	7150374400	6515	7.1
16000	7295	7335073998	6028	3.2
32000	7442	7501421998	5949	4.0
48000	7689	7753116800	5925	3.6
64000	7904	7989797999	5935	4.4
80000	8124	8213160600	5839	3.9
160000	9943	10090948000	6472	4.3
240000	11569	11772727399	6859	4.3
320000	11792	12048516799	5832	4.3
400000	13493	13811299600	6371	4.5

Table 5.4.2 The results from running the ping-pong tests on 3 topologies with different message sizes.

5.5 Running and Simulating NASPB

The NASA Advanced Supercomputing (NAS) Parallel Benchmarks (NASPB or NPB) are a small set of programs designed to evaluate the performance of parallel supercomputers (or a relatively small cluster as in this master thesis). The benchmarks are derived from computational fluid dynamics applications. [33]

NASPB resemble the MPI applications that are typically run on computer clusters. This is the type of applications the simulation setup of this master thesis aims at simulating. Naturally there are numerous other MPI applications / benchmarks, for example Sweep3D[2] or HPCC[3]. However, there are reasons why I chose not to use these two benchmarks. Sweep3D produces too little network traffic. The LogGOPSim part of the HPCC simulation could not terminate correctly (probably due to a bug in the Schedgen; the numbers of sends and receives did not match). Understanding the problem would require reading of the MPI traces collected from an HPCC run; however, these traces were too large (over 100MB) to read manually, so it was considered inappropriate to spend time on this.

Simulating NASPB is done to test the behavior and correctness of the simulator after it had been calibrated using the small and transparent MPI applications. The MPI traces from NASPB runs are relatively large consisting of thousands of lines each, and are thus hard, if not impossible to read manually. So we use only the matching between the real final running time of a NASPB program and the corresponding simulated time to evaluate the correctness of the simulation setup.

There are 8 small programs in the benchmark set: [34]

- EP (embarrassingly parallel)
- MG (multigrid)
- CG (conjugate gradient)
- FT (Fast Fourier Transform)
- IS (integer sort)
- LU (lower and upper triangular system solution)
- SP (scalar pentadiagonal equations)
- BT (block tridiagonal equations)

Each of the programs listed above comes in several sizes (classes): A, B, C, D, W(orkstation) and S(ample). The 'S' problem size is the smallest, and 'D' is the largest. For example on four cluster nodes the SP program of class 'S' takes 0.185 seconds to run, while the runtime of class 'A' is 53 seconds, and the runtime of class 'B' is 223 seconds.

The details of what these programs are doing are not relevant for this master thesis. It is important to mention that only 5 out of 8 will be used. The first of the 3 omitted programs, FT, did not compile like all the others and it was considered inappropriate to use too much effort on attempting to compile it. The other one, EP, barely uses interprocessor communication, and thus is not interesting for us. The third one, IS, uses MPI_Alltoallv function which is not implemented in Schedgen1.1. However the remaining 5 programs: SP, MG, CG, LU and BT should be enough to run some tests and see whether the simulation makes sense. Only the 'S' problem sizes will be used due to scalability of the simulation – the longer the simulated time, the longer the simulation runtime is.

The five NASPB programs have been run first on the cluster at Simula Research Laboratory, and then simulated using the LogGOPSim & IB model in Omnet++ integration. The topology used during all tests was fat tree with 8 end nodes (H8_S6) shown on Figure 5.1.

Three out of five benchmarks require square number of nodes (1, 4, 9...). All five tests have been

2 http://wwwc3.lanl.gov/pal/software/sweep3d/
3 http://icl.cs.utk.edu/hpcc/

run using 4 nodes (the ones marked H_1 to H_4 on Figure 5.3). The two programs that don't require square number of nodes (MG and CG) were tested also on 8 nodes.

5.5.1 Simulating NASPB With Processing Overheads Approach

First we simulate the NASPB programs using the processing overheads approach. The latest version of LogGOPSim & IB model in Omnet++ integration was used, i.e. with all the "enhancements": pinning delays and two sets of parameters. The parameters were:

- processing overhead per message for small messages: 25,979,800 picoseconds
- processing overhead per byte for small messages: 155 picoseconds
- processing overhead per message for large messages: 32,000,000 picoseconds
- processing overhead per byte for large messages: 65 picoseconds

All the other parameters (link latency and network overheads) were set to zero. Tables 5.5.1.1 and 5.5.1.2 show the simulated running times (of the first node), real running times calculated based on the trace file, the difference between these two times in percent (the real running time taken as base) and finally the time it took to run each simulation (this one is included just to provide some extra insight into the efficiency of the integration). All simulated running times and real running times are given in picoseconds, the test times are given in seconds (ranging between 12.5 and 62 minutes).

	Simulated total run time [ps]	Real runtime [ps]	Deviation (real as base)	Simulation runtime [s]
SP	188789626550	185470000000	1.76	2410
BT	205686594900	217644000000	-5.81	2594
MG	66955150080	62737000000	6.30	750
LU	146306189920	140837000000	3.74	1756
CG	230182120620	228983000000	0.52	2939

Table 5.5.1.1 NASPB test results for 4 nodes.

	Simulated total run time [ps]	Real runtime [ps]	Deviation (real as base)	Simulation runtime [s]
MG	68015699520	69630000000	-2.37	751
CG	297888068480	286205000000	3.92	3719

Table 5.5.1.2 NASPB test results for 8 nodes.

We can see that the difference between the real and simulated running times ranges between 0.5 and 6.3%. One explanation for this error could be an error in the LogGOPSim parameters deduced using linear regression. Another explanation could be the collective operations which are simulated with smaller precision than point-to-point operations. The overwhelming majority of operations in the NASPB programs are point-to-point, which can explain the relatively small error size.

5.5.2 Simulating NASPB With MPI Function Durations Approach

The NASPB programs were also simulated using the MPI function durations approach. The results from the 7 NASPB tests are presented in Table 5.5.2.1 and 5.5.2.2. As we can see the error increases significantly with the number of participating nodes. The more nodes are participating – the larger the chance that dis-synchronization takes place. We can also see that the errors are generally larger than the ones during simulations of the ping-pong application.

	Simulated total run time [ps]	Real runtime [ps]	Difference [%]
LU	168765763054	140837000000	19.83
CG	259766287500	228983000000	13.44
BT	242301671050	217644000000	11.33
SP	203899267399	185470000000	9.94
MG	74633421850	62737000000	18.96

Table 5.5.2.1 The results from running the NASPB tests on 4 nodes.

	Simulated total run time [ps]	Real runtime [ps]	Difference [%]
CG	405502469700	228983000000	77.09
MG	130785038196	62737000000	108.47

Table 5.5.2.2 The results from running the NASPB tests on 8 nodes.

5.6 Conclusion

As we could see from the test results, using the LogGOPS processing overhead parameters derived from point to point operations gives more precise results for most MPI applications. Using function duration delays from the traces as local calculations surprisingly enough leads to quite considerable deviations between the real and the simulated total running times. This is caused by the “dis-synchronization” between the simulated ranks which often causes that some ranks have to wait for the other, which did not take place in the real world. The solution to this problem is quite challenging, fortunately we can always fall back to the processing overheads solution, which gives good enough results.

Chapter 6 Conclusion

6.1 Related Work

There are several models and simulators for simulating HPC systems. In this section I will briefly describe a few simulators which have features common with the integration of the IB Model and LogGOPSim presented in this thesis.

The MARS (MPI Application Replay network Simulator) is the one that resembles my work most. The MARS framework is described in [45]. It is trace-driven and uses the Omnest simulation framework, which is largely identical to Omnet++. Similarly to the IB Model simulation, a MARS simulation consists of modules representing the network elements, like switches and network adapters. It also contains the processing node modules serving as sources of network traffic with the pattern based on the MPI traces.

Another simulator that rebuilds the behavior of a parallel program from a set of traces is Dimemas [section 4 in 46]. A simple linear latency and bandwidth model is used in Dimemas to simulate the message traveling time (the possible conflicts between the packets is taken into account for bus model networks though, i.e. when two packets cannot be sent simultaneously). In other words the network is modeled at a high abstraction level. The processing node hardware is also been simulated using several parameters such as processor speed and scheduling policy (for simulating multi-threaded applications), communication latency between processors within the same node, etc. Dimemas produces results in the form of another trace file.

Edinet presented in [47] is an Execution Driven Interconnection Network simulator for distributed shared memory systems. Edinet consists of two simulators, just like my integration of the IB Model and LogGOPSim in this master thesis. One part of Edinet is an execution driven simulator modeling the memory subsystem. In execution driven simulators "*an application runs on the host processor and special call-outs are inserted into the original code to instrument the required events. These events are scheduled as requests to the simulator.* " [47] In the case of Edinet these events are send/receive requests, and the simulator is an interconnection network simulator (the second simulator Edinet consists of.)

Naturally, there are several similarities between my integration and the simulators described above. Although my integration consists of two relatively separate simulators just like Edinet, my simulation setup is trace-driven, while Edinet is execution-driven. The trace-driven MARS framework utilizes an approach which is somewhat similar to my MPI function durations approach described in Section 5.4, while I decided to use the processing overheads of the LogGOPS model. Finally my simulation setup has a lower abstraction level (more detailed) than Dimemas, although it is hard to say whether it is a benefit or a drawback – lower abstraction levels generally come at a price of efficiency.

6.2 Conclusion

The main goal of this master thesis was introducing means of using real life network traffic patterns in the given simulation environment – the IB Model in Omnet++. To achieve this goal an integration of two simulators has been implemented. The simulators are LogGOPSim, responsible for producing real life network traffic based on the MPI traces, and the IB Model in

Omnet++, simulating the Link and Physical Layers.

The implementation has been done using Linux interprocess communication. It was very convenient that the two integrated simulators had modular structure because adding new or substituting the existing modules was the essential method of integration. The Network module of LogGOPSim has been substituted by another one and the Generator simple module has to be part of all the simulated Infiniband networks in Omnet++. One of the greatest challenges experienced during the implementation was ineffectiveness of integration if preserving the LogGOPSim core and its Network module interface intact. The problem was solved by changing the core module of LogGOPSim, which opened the way for making the Network module of LogGOPSim much simpler. The last version of the integration is relatively effective – it is the IB Model in Omnet++ that is the bottleneck.

The integration alone was not enough to provide the realistic network traffic pattern for the simulation. LogGOPSim simulation based on the schedule produced by the standard Schedgen1.1 reflects only what is on top of MPI, while the IB Model in Omnet++ simulates only what is on Link Layer and below. The Network and Transport layers had to be simulated too somehow. I have investigated two different approaches to solve this. The first approach involved using the parameters of LogGOPSim to compensate for the time messages spend traveling through the Transport and Network layers. The other approach, seeming very promising at first, was about using function duration times taken directly from the MPI trace to compensate for Network and Transport layer delays. The other approach involved doing some changes to Schedgen1.1. It was the first approach that turned out to give the best results, so the final choice is to stick to this first approach, although it is not flawless either.

6.3 Future Work

The first thing I can think of when it comes to possible future work is trying to put the source code of LogGOPSim into the Generator simple module of the network simulated in Omnet++. This way it won't be an integration of two separate simulators, but rather one-piece simulator. One benefit of this is that the simulator would be more user friendly and probably slightly more effective when it comes to performance. From the other side it would be more difficult to make any likely updates of the real LogGOPSim to this simulator.

The other thing that can be done concerns the second approach to compensating the delays of the Network and Transport layers. That approach involves using the MPI function delays from the traces. It can potentially give very precise results, and the simulated network traffic will reflect the real life network traffic in a more precise way. To further develop this approach one would need to cardinally change the way Schedgen works. The multiple trace-files have to be parsed simultaneously (now they are parsed sequentially) and all the send and receive operations have to be inserted into the .goal schedule in such a way that they happen approximately at the same time.

Additionally the Schedgen does not support the conversion of several MPI function calls from a trace into a .goal schedule. This means that an MPI application using a function which was not implemented in Schedgen cannot be simulated.

One can also think of making the IB Model in Omnet++ distributed, which will probably improve the performance. However, this task is not trivial.

Another thing that can be done is a detailed study of the network traffic produced by the MPI applications and the possibility of synthesizing it in a simulation, i.e. without using the traces.

References

[1] Descrete-Event System Simulation, third edition, Jerry Banks et al., Prentice Hall, 2001

[2] Simulation versus Analytic Modeling in Large Computing Environments, Dr. Bernie Domanski, Responsive Systems Company, 1999

[3] Explanatory & Analytical Models: http://faculty.washington.edu/krumme/207/models.html

[4] Network Simulation Definition: http://en.wikipedia.org/wiki/Network_simulation 15.03.2011

[5] Omnet++ User Manual version 4.1

[6] InfiniBand Architecture Specification Volume 1 Release 1.2.1, InfiniBand Trade Association, 2007

[7] LogGPS: A Parallel Computational Model for Synchronization Analysis, F. Ino, N. Fujimoto, and K. Hagihara, ACM Symposium on Principles and Practices of Parallel Programming, 2001

[8] LogGOPSim - Simulating Large-Scale Applications in the LogGOPS Model, T. Hoefler, T. Schneider and A. Lumsdaine, Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, 2010

[9] Network Congestion Definition: http://www.linfo.org/congestion.html

[10] http://en.wikipedia.org/wiki/Ethernet_flow_control 06.04.2011

[11] InfiniBand Congestion Control Modelling and validation, Ernst Gunnar Gran, Sven-Arne Reinemo, 4th International ICST Conference on Simulation Tools and Techniques (SIMUTools2011, OMNeT++ 2011 Workshop), 2011

[12] Blaise Barney, Lawrence Livermore National Laboratory: Introduction to Parallel Computing

https://computing.llnl.gov/tutorials/parallel_comp/

[13] Blaise Barney, Lawrence Livermore National Laboratory: OpenMp https://computing.llnl.gov/tutorials/openMP/

[14] OpenMp A Parallel Programming Model for Shared Memory Architectures, Paul Graham, Edinburgh Parallel Computing Center, March 1999

[15] A. D. Marshall 1994-2005 - IPC:Shared Memory http://www.cs.cf.ac.uk/Dave/C/node27.html

[16] IPC: Message Queues http://www.cs.cf.ac.uk/Dave/C/node25.html

[17] Erlang: http://www.erlang.org/

[18] MPI-2: Extensions to the Message-Passing Interface: http://www.mcs.anl.gov/research/projects/mpi/mpi-standard/mpi-report-2.0-sf/mpi2-report.htm

[19] MPI: A message-passing interface standard. Technical Report UT- CS-94-230 1994

[20] MPI_Alltoall function: http://mpi.deino.net/mpi_functions/MPI_Alltoall.html

[21] Ron Brightwell et. Al: A Preliminary Analysis of the MPI Queue Characterisitics of Several Applications (about receive and unexpected queues)

https://computing.llnl.gov/tutorials/mpi/

[22] Eager messages: http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp?topic=%2Fcom.ibm.cluster.pe432.mpiprog.doc%2Fam106_eagermess.html

[23] MPI Performance analysis tools: http://www.open-mpi.org/faq/?category=perftools

[24] Profiling: http://en.wikipedia.org/wiki/Profiling_%28computer_programming%29 07.04.2011

[25] Efficient Program Tracing by James R. Larus, University of Wisconsin-Madison, 1993

[26] Packet tracing: http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp?topic=/com.ibm.aix.commadmn/doc/commadmndita/packet_tracing.htm

[27] Automated Packet Trace Analysis of TCP Implementations by Vern Paxson, Network Research Group Lawrence Berkley National Laboratory, University of California, Berkley, 1997

[28] Group Operation Assembly Language : www.unixer.de/publications/img/hoefler-goal-slides.pdf

[29] Infiniband model in Omnet++ source code

[30] Flit-Reservation Flow Control, Li-Shiuan Peh and William J. Dally, Proceedings of the 6th International Symposium on High-Performance Computer Architecture , 2000

[31] FAQ: Compiling MPI applications: http://www.open-mpi.org/faq/?category=mpi-apps#static-mpi-apps

[32] Linear Regression implementation by David C. Swaim: http://david.swaim.com/cpp/linreg.htm

[33] NAS Parallel Benchmarks: http://www.nas.nasa.gov/Resources/Software/npb.html

[34] The NAS Parallel Benchmarks by D. H. Bailey et al., NASA Ames Research Center , 2000

[35] Host Side Dynamic Reconfiguration with InfiniBand, Wei Lin Guay et al., 2010 IEEE International Conference on Cluster Computing, ed. by Xiaohui Gu and Xiaosong Ma, pp. 126-135, IEEE Computer Society (ISBN: 978-0-7695-4220-1), 2010

[36] An Introduction to the InfiniBandTM Architecture (http://www.infinibandta.org) , Gregory F. Pfister, IEEE Press, 2001

[37] Advanced C Programming. Profiling. Sebastian Hack http://www.mpi-inf.mpg.de/departments/rg1/teaching/advancedc-ws08/script/lecture06.pdf

[38] The World as a Process: Simulations in the Natural and Social Sciences, Stephan Hartmann in R. Hegselmann, U. Mueller and K. Troitzsch, eds. Modelling and Simulation in the Social Sciences from the Philosophy of Science Point of View. Dordrecht: Kluwer Academic Publishers, pp. 77-100 , 1996

[39] Continuous Simulation, Wouter Duivesteijn (slides), 2006

[40] Infiniband FAQ: www.mellanox.com/pdf/whitepapers/InfiniBandFAQ_FQ_100.pdf

[41] Trace-driven Co-simulation of High-Performance Computing Systems using OMNeT++, Cyriel Minkenberg and Germán Rodriguez Herrera, Proc. 2Nd International Workshop on OMNeT++, held in conjuction with the Second International Conference on Simulation Tools

and Techniques (SIMUTools'09), 2009.

[42] Interconnection Networks, José Duato et al., Morgan Kaufmann Publishers, 2003

[43] Table of Infiniband link speeds: http://en.wikipedia.org/wiki/InfiniBand#Description 22.08.2011

[44] Wikipedia article about concurrent computing: http://en.wikipedia.org/wiki/Concurrent_computing#Concurrent_programming_languages 22.07.2011

[45] A Framework for End-to-end Simulation of High-performance Computing Systems, Wolfgang E. Denzel et al., presented at SIMUTools'08, March 2008

[46] DiP : A Parallel Program Development Environment , Jesus Labarta et al., Euro-Par'96, Lyon, France, August 1996

[47] Edinet: An Execution Driven Interconnection Network Simulator for DSM Systems, J. Flich et al., International Conference on Parallel Processing, 1999

Appendixes

Appendix A

Pseudo-code event handlers for the example used to explain the discrete-event simulations (traffic light).

```
Arrival event handler:
add new arrival event with time stamp now + 5 seconds
if (green and waiting == 0 and last departure event occurred at least
2 seconds ago):
      add new departure event with time stamp now
else if (green and waiting == 0):
      add new departure event with time stamp now+2-now-time for last
            departure event
else:
      increment waiting counter
Departure event handler:
if (red):
      increment waiting counter
else if (green and last departure event occurred at least 2 seconds
ago):
      decrement waiting counter (not past 0)
      if (waiting counter > 0):
            add new departure event with time stamp now+2 seconds
else:
      add new departure event with time stamp now+2-now-time for last
      departure event
Red to green light transition event handler:
      change the traffic light state variable
      add new green to red transition event with time stamp now+19
seconds
      if (waiting > 0 and last departure occurred at least 2 seconds
ago):                  add new departure event with time stamp now
      else if (waiting > 0):
            add new departure event with time stamp now+2-now-time for
last            departure event
Green to red light transition event handler:
      change the traffic light state variable
      add red to green transition event with time stamp now+19 seconds
```

Appendix B

Source code of a small MPI application “ltest” used for illustration of what's going on.

```
#include <mpi.h>
#include <stdio.h>

#define DATASIZE 100000
```

```
int main(int argc, char **argv) {
  int rank;
  int procs; //number of mpi procs
  int i[DATASIZE]; //just some buffer
  int ret; //just a counter
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &procs);

  if (rank == 0) { //what the first node is doing
    for(ret=0; ret<10; ret++) {
      MPI_Send(i, DATASIZE, MPI_INT, 1, 0, MPI_COMM_WORLD);
      MPI_Recv(i, DATASIZE, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
    }
      printf("%d done\n", rank);
  }
  else if (rank == 1) { //what the second node is doing
    for(ret=0; ret<10; ret++) {
      MPI_Recv(i, DATASIZE, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
      MPI_Send(i, DATASIZE, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
      printf("%d done\n", rank);
  }

  fflush(stdout);

  MPI_Finalize();

  return 0;
}
```

Appendix C

Trace file for the first node:

```
# htor's PMPI Tracer 0.9 Output File
# time: Thu Apr 14 09:49:34 2011
# hostname: compute-0-13.local.(none)
# uname: Linux compute-0-13.local 2.6.18-164.6.1.el5 #1 SMP Tue Nov 3 16:12:36 EST 2009 x86_64
# clockdiff: 0.000000 s (relative to rank 0)

MPI_Init:-:140735615211452:140735615211440:1302767374553965
MPI_Comm_rank:1302767374553992:6575584,0,2:140735615611496:1302767374554004
MPI_Comm_size:1302767374554016:6575584,0,2:140735615611492:1302767374554026
MPI_Send:1302767374554038:140735615211488:100000:1,4,4:1:0:6575584,0,2:1302767374555673
MPI_Recv:1302767374555688:140735615211488:100000:1,4,4:1:0:6575584,0,2:140735615211456:130276737455
5978
MPI_Send:1302767374555987:140735615211488:100000:1,4,4:1:0:6575584,0,2:1302767374556261
MPI_Recv:1302767374556270:140735615211488:100000:1,4,4:1:0:6575584,0,2:140735615211456:130276737455
6538
MPI_Send:1302767374556546:140735615211488:100000:1,4,4:1:0:6575584,0,2:1302767374556820
MPI_Recv:1302767374556829:140735615211488:100000:1,4,4:1:0:6575584,0,2:140735615211456:130276737455
7095
MPI_Send:1302767374557104:140735615211488:100000:1,4,4:1:0:6575584,0,2:1302767374557378
MPI_Recv:1302767374557387:140735615211488:100000:1,4,4:1:0:6575584,0,2:140735615211456:130276737455
```

7654
MPI_Send:1302767374557663:140735615211488:100000:1,4,4:1:0:6575584,0,2:1302767374557936
MPI_Recv:1302767374557945:140735615211488:100000:1,4,4:1:0:6575584,0,2:140735615211456:130276737455
8211
MPI_Send:1302767374558220:140735615211488:100000:1,4,4:1:0:6575584,0,2:1302767374558493
MPI_Recv:1302767374558502:140735615211488:100000:1,4,4:1:0:6575584,0,2:140735615211456:130276737455
8768
MPI_Send:1302767374558777:140735615211488:100000:1,4,4:1:0:6575584,0,2:1302767374559049
MPI_Recv:1302767374559058:140735615211488:100000:1,4,4:1:0:6575584,0,2:140735615211456:130276737455
9325
MPI_Send:1302767374559334:140735615211488:100000:1,4,4:1:0:6575584,0,2:1302767374559606
MPI_Recv:1302767374559615:140735615211488:100000:1,4,4:1:0:6575584,0,2:140735615211456:130276737455
9881
MPI_Send:1302767374559890:140735615211488:100000:1,4,4:1:0:6575584,0,2:1302767374560162
MPI_Recv:1302767374560170:140735615211488:100000:1,4,4:1:0:6575584,0,2:140735615211456:130276737456
0445
MPI_Send:1302767374560454:140735615211488:100000:1,4,4:1:0:6575584,0,2:1302767374560728
MPI_Recv:1302767374560737:140735615211488:100000:1,4,4:1:0:6575584,0,2:140735615211456:130276737456
1003
Finalize clockdiff: 0.000000
MPI_Finalize:1302767374567772:-

Trace file for the second node:

htor's PMPI Tracer 0.9 Output File
time: Thu Apr 14 09:49:34 2011
hostname: compute-0-6.local.(none)
uname: Linux compute-0-6.local 2.6.18-164.6.1.el5 #1 SMP Tue Nov 3 16:12:36 EST 2009 x86_64
clockdiff: -289.201736 s (relative to rank 0)

MPI_Init:-:140736717160604:140736717160592:1302767374554203
MPI_Comm_rank:1302767374554225:6575584,1,2:140736717560648:1302767374554238
MPI_Comm_size:1302767374554250:6575584,1,2:140736717560644:1302767374554260
MPI_Recv:1302767374554273:140736717160640:100000:1,4,4:0:0:6575584,1,2:140736717160608:13027673745
55960
MPI_Send:1302767374555975:140736717160640:100000:1,4,4:0:0:6575584,1,2:1302767374556269
MPI_Recv:1302767374556278:140736717160640:100000:1,4,4:0:0:6575584,1,2:140736717160608:13027673745
56550
MPI_Send:1302767374556559:140736717160640:100000:1,4,4:0:0:6575584,1,2:1302767374556828
MPI_Recv:1302767374556837:140736717160640:100000:1,4,4:0:0:6575584,1,2:140736717160608:13027673745
57108
MPI_Send:1302767374557117:140736717160640:100000:1,4,4:0:0:6575584,1,2:1302767374557385
MPI_Recv:1302767374557395:140736717160640:100000:1,4,4:0:0:6575584,1,2:140736717160608:13027673745
57665
MPI_Send:1302767374557674:140736717160640:100000:1,4,4:0:0:6575584,1,2:1302767374557942
MPI_Recv:1302767374557951:140736717160640:100000:1,4,4:0:0:6575584,1,2:140736717160608:13027673745
58223
MPI_Send:1302767374558232:140736717160640:100000:1,4,4:0:0:6575584,1,2:1302767374558501
MPI_Recv:1302767374558510:140736717160640:100000:1,4,4:0:0:6575584,1,2:140736717160608:13027673745
58780
MPI_Send:1302767374558789:140736717160640:100000:1,4,4:0:0:6575584,1,2:1302767374559057
MPI_Recv:1302767374559066:140736717160640:100000:1,4,4:0:0:6575584,1,2:140736717160608:13027673745
59338
MPI_Send:1302767374559347:140736717160640:100000:1,4,4:0:0:6575584,1,2:1302767374559615
MPI_Recv:1302767374559624:140736717160640:100000:1,4,4:0:0:6575584,1,2:140736717160608:13027673745
59895

MPI_Send:1302767374559904:140736717160640:100000:1,4,4:0:0:6575584,1,2:1302767374560172
MPI_Recv:1302767374560181:140736717160640:100000:1,4,4:0:0:6575584,1,2:140736717160608:1302767374560451
MPI_Send:1302767374560460:140736717160640:100000:1,4,4:0:0:6575584,1,2:1302767374560733
MPI_Recv:1302767374560742:140736717160640:100000:1,4,4:0:0:6575584,1,2:140736717160608:1302767374561015
MPI_Send:1302767374561024:140736717160640:100000:1,4,4:0:0:6575584,1,2:1302767374561292
Finalize clockdiff: -288.486481
MPI_Finalize:1302767374568092:-

Appendix D

The .goal file produced from the traces of running the small MPI test program:

```
num_ranks 2

rank 0 {
l1: send 400000b to 1 tag 0
l2: calc 73000000
l1 requires l2
l3: recv 400000b from 1 tag 0
l4: calc 15000000
l3 requires l4
l4 requires l1
l5: send 400000b to 1 tag 0
l6: calc 9000000
l5 requires l6
l6 requires l3
l7: recv 400000b from 1 tag 0
l8: calc 9000000
l7 requires l8
l8 requires l5
l9: send 400000b to 1 tag 0
l10: calc 8000000
l9 requires l10
l10 requires l7
l11: recv 400000b from 1 tag 0
l12: calc 9000000
l11 requires l12
l12 requires l9
l13: send 400000b to 1 tag 0
l14: calc 9000000
l13 requires l14
l14 requires l11
l15: recv 400000b from 1 tag 0
l16: calc 9000000
l15 requires l16
l16 requires l13
l17: send 400000b to 1 tag 0
l18: calc 9000000
l17 requires l18
l18 requires l15
l19: recv 400000b from 1 tag 0
l20: calc 9000000
l19 requires l20
l20 requires l17
l21: send 400000b to 1 tag 0
```

l22: calc 9000000
l21 requires l22
l22 requires l19
l23: recv 400000b from 1 tag 0
l24: calc 9000000
l23 requires l24
l24 requires l21
l25: send 400000b to 1 tag 0
l26: calc 9000000
l25 requires l26
l26 requires l23
l27: recv 400000b from 1 tag 0
l28: calc 9000000
l27 requires l28
l28 requires l25
l29: send 400000b to 1 tag 0
l30: calc 9000000
l29 requires l30
l30 requires l27
l31: recv 400000b from 1 tag 0
l32: calc 9000000
l31 requires l32
l32 requires l29
l33: send 400000b to 1 tag 0
l34: calc 9000000
l33 requires l34
l34 requires l31
l35: recv 400000b from 1 tag 0
l36: calc 8000000
l35 requires l36
l36 requires l33
l37: send 400000b to 1 tag 0
l38: calc 9000000
l37 requires l38
l38 requires l35
l39: recv 400000b from 1 tag 0
l40: calc 9000000
l39 requires l40
l40 requires l37
l41: calc 6769000000
l41 requires l39
}

rank 1 {
l1: recv 400000b from 0 tag 0
l2: calc 70000000
l1 requires l2
l3: send 400000b to 0 tag 0
l4: calc 15000000
l3 requires l4
l4 requires l1
l5: recv 400000b from 0 tag 0
l6: calc 9000000
l5 requires l6
l6 requires l3
l7: send 400000b to 0 tag 0

l8: calc 9000000
l7 requires l8
l8 requires l5
l9: recv 400000b from 0 tag 0
l10: calc 9000000
l9 requires l10
l10 requires l7
l11: send 400000b to 0 tag 0
l12: calc 9000000
l11 requires l12
l12 requires l9
l13: recv 400000b from 0 tag 0
l14: calc 10000000
l13 requires l14
l14 requires l11
l15: send 400000b to 0 tag 0
l16: calc 9000000
l15 requires l16
l16 requires l13
l17: recv 400000b from 0 tag 0
l18: calc 9000000
l17 requires l18
l18 requires l15
l19: send 400000b to 0 tag 0
l20: calc 9000000
l19 requires l20
l20 requires l17
l21: recv 400000b from 0 tag 0
l22: calc 9000000
l21 requires l22
l22 requires l19
l23: send 400000b to 0 tag 0
l24: calc 9000000
l23 requires l24
l24 requires l21
l25: recv 400000b from 0 tag 0
l26: calc 9000000
l25 requires l26
l26 requires l23
l27: send 400000b to 0 tag 0
l28: calc 9000000
l27 requires l28
l28 requires l25
l29: recv 400000b from 0 tag 0
l30: calc 9000000
l29 requires l30
l30 requires l27
l31: send 400000b to 0 tag 0
l32: calc 9000000
l31 requires l32
l32 requires l29
l33: recv 400000b from 0 tag 0
l34: calc 9000000
l33 requires l34
l34 requires l31
l35: send 400000b to 0 tag 0

```
l36: calc 9000000
l35 requires l36
l36 requires l33
l37: recv 400000b from 0 tag 0
l38: calc 9000000
l37 requires l38
l38 requires l35
l39: send 400000b to 0 tag 0
l40: calc 9000000
l39 requires l40
l40 requires l37
l41: calc 6800000000
l41 requires l39
}
```

Appendix E

The two log files produced during simulation of the small test program.

The inserts file:

```
insert handle: 0 simTime(): 0 currtime_l: 73000000 src_l: 0 dest_l: 1
insert handle: 1 simTime(): 0.00029128685 currtime_l: 361286850 src_l: 1 dest_l: 0
insert handle: 2 simTime(): 0.00057885325 currtime_l: 642853250 src_l: 0 dest_l: 1
insert handle: 3 simTime(): 0.0008611345 currtime_l: 925134500 src_l: 1 dest_l: 0
insert handle: 4 simTime(): 0.00114327825 currtime_l: 1206278250 src_l: 0 dest_l: 1
insert handle: 5 simTime(): 0.0014245306 currtime_l: 1488530600 src_l: 1 dest_l: 0
insert handle: 6 simTime(): 0.00170619075 currtime_l: 1770190750 src_l: 0 dest_l: 1
insert handle: 7 simTime(): 0.0019883345 currtime_l: 2052334500 src_l: 1 dest_l: 0
insert handle: 8 simTime(): 0.00227044075 currtime_l: 2334440750 src_l: 0 dest_l: 1
insert handle: 9 simTime(): 0.002552647 currtime_l: 2616647000 src_l: 1 dest_l: 0
insert handle: 10 simTime(): 0.0028348181 currtime_l: 2898818100 src_l: 0 dest_l: 1
insert handle: 11 simTime(): 0.003116447 currtime_l: 3180447000 src_l: 1 dest_l: 0
insert handle: 12 simTime(): 0.0033987556 currtime_l: 3462755600 src_l: 0 dest_l: 1
insert handle: 13 simTime(): 0.0036803845 currtime_l: 3744384500 src_l: 1 dest_l: 0
insert handle: 14 simTime(): 0.0039627345 currtime_l: 4026734500 src_l: 0 dest_l: 1
insert handle: 15 simTime(): 0.004244772 currtime_l: 4308772000 src_l: 1 dest_l: 0
insert handle: 16 simTime(): 0.00452704935 currtime_l: 4591049350 src_l: 0 dest_l: 1
insert handle: 17 simTime(): 0.004808647 currtime_l: 4872647000 src_l: 1 dest_l: 0
insert handle: 18 simTime(): 0.00509092825 currtime_l: 5154928250 src_l: 0 dest_l: 1
insert handle: 19 simTime(): 0.00537324935 currtime_l: 5437249350 src_l: 1 dest_l: 0
```

The arrivals file:

```
sink: message 0 has arrived at 0.00029128685
sink: message 1 has arrived at 0.00057885325
sink: message 2 has arrived at 0.0008611345
sink: message 3 has arrived at 0.00114327825
sink: message 4 has arrived at 0.0014245306
sink: message 5 has arrived at 0.00170619075
sink: message 6 has arrived at 0.0019883345
sink: message 7 has arrived at 0.00227044075
sink: message 8 has arrived at 0.002552647
sink: message 9 has arrived at 0.0028348181
sink: message 10 has arrived at 0.003116447
sink: message 11 has arrived at 0.0033987556
sink: message 12 has arrived at 0.0036803845
```

```
sink: message 13 has arrived at 0.0039627345
sink: message 14 has arrived at 0.004244772
sink: message 15 has arrived at 0.00452704935
sink: message 16 has arrived at 0.004808647
sink: message 17 has arrived at 0.00509092825
sink: message 18 has arrived at 0.00537324935
sink: message 19 has arrived at 0.005654847
```

Appendix F

The algorithm for detecting buffer overlap:

```
#include <set>
/** Inserts buffers/intervals into a datastructure.
  * If the new buffer overlaps with an old one, they are merged.
  * Returns the number of non overlapping bytes.
  */
std::set<std::pair<unsigned long long int, unsigned long long int> > bufs;
unsigned long long int insert_buf(unsigned long long int start_addr, unsigned long long int end_addr) {
  std::set<std::pair<unsigned long long int, unsigned long long int> >::iterator it, it3;
  std::pair<std::set<std::pair<unsigned long long int, unsigned long long int> >::iterator, bool> it2;
  unsigned long long int old_buf_start, old_buf_end, ret_size=0;
  if(start_addr > end_addr) std::cout << "What do you think you're doing!?" << std::endl;

  for (it = bufs.begin(); it != bufs.end(); it++) {
    old_buf_start = it->first;
    old_buf_end = it->second;
    if(start_addr <= old_buf_end && end_addr >= old_buf_start) { //overlap with one other buffer
      bufs.erase(it);
      it2 = bufs.insert(std::make_pair(std::min(old_buf_start, start_addr),
                                       std::max(old_buf_end, end_addr)));

      //calculate the non-overlapping size
      if (old_buf_start > start_addr) ret_size += old_buf_start - start_addr;
      if (old_buf_end < end_addr) ret_size += end_addr - old_buf_end;

      //need to check for new partial (full not possible) overlaps between the new entry and the existing
ones and merge if necessary
      start_addr= it2.first->first;
      end_addr = it2.first->second;
      for (it3 = bufs.begin(); it3 != bufs.end(); it3++) { //for each element except the new one
            if(it3 == it2.first) continue;
            old_buf_start = it3->first;
            old_buf_end = it3->second;
            if(start_addr <= old_buf_end && end_addr >= old_buf_start) { //overlap with one other buffer
              bufs.erase(it3);
              bufs.erase(it2.first);
              bufs.insert(std::make_pair(std::min(old_buf_start, start_addr),
                                       std::max(old_buf_end, end_addr)));
              //need to decrement the ret_size with the size of the newly found overlap...
              if (old_buf_start >= start_addr) ret_size -= std::min(end_addr, old_buf_end) - old_buf_start;
              else if (old_buf_end <= end_addr) ret_size -= old_buf_end - std::max(start_addr,
old_buf_start);
            }
      }

      return ret_size;
    }
  } //end of main for-loop
```

```
  //no overlaps found, just insert...
  bufs.insert(std::make_pair(start_addr, end_addr));
  return end_addr - start_addr;
}

/** Delete all the elements in the bufs set */
void pin_flush() {
  bufs.clear();
}
```

Appendix G

A simple MPI application heavily using collective operations:

```
#include <mpi.h>
#include <stdio.h>

#define DATASIZE 100000

int main(int argc, char **argv) {
  int rank; //id of this proc
  int procs; //number of mpi procs
  int i[DATASIZE]; //just some buffers
  int f[DATASIZE];
  int ret; //just a counter
  MPI_Status status;
i[0] = 6;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &procs);

  if (!(rank%2)) { //what the first node is doing
          MPI_Send(f, DATASIZE, MPI_INT, rank+1, 0, MPI_COMM_WORLD);
          MPI_Recv(f, DATASIZE, MPI_INT, rank+1, 0, MPI_COMM_WORLD, &status);
          MPI_Send(i, DATASIZE, MPI_INT, rank+1, 0, MPI_COMM_WORLD);
          MPI_Recv(i, DATASIZE, MPI_INT, rank+1, 0, MPI_COMM_WORLD, &status);
  }
  else { //what the second node is doing
          MPI_Recv(f, DATASIZE, MPI_INT, rank-1, 0, MPI_COMM_WORLD, &status);
          MPI_Send(f, DATASIZE, MPI_INT, rank-1, 0, MPI_COMM_WORLD);
          MPI_Recv(i, DATASIZE, MPI_INT, rank-1, 0, MPI_COMM_WORLD, &status);
          MPI_Send(i, DATASIZE, MPI_INT, rank-1, 0, MPI_COMM_WORLD);
  }

  MPI_Barrier(MPI_COMM_WORLD);
  for (ret=1; ret<=10; ret++) {
          MPI_Allgather(i, ret*300, MPI_INT, f, ret*300, MPI_INT, MPI_COMM_WORLD);
          MPI_Barrier(MPI_COMM_WORLD);
          MPI_Allreduce(i, f, ret*600, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
          MPI_Barrier(MPI_COMM_WORLD);
  }

printf("%d done %d\n", rank, f[0]);

  fflush(stdout);
```

```
  MPI_Finalize();

  return 0;
}
```

Appendix H

The trace collected from the first cluster node running the test application heavily using the MPI collective operations.

```
# htor's PMPI Tracer 0.9 Output File
# time: Wed May 18 09:26:05 2011
# hostname: compute-0-13.local.(none)
# uname: Linux compute-0-13.local 2.6.18-164.6.1.el5 #1 SMP Tue Nov 3 16:12:36 EST 2009 x86_64
# clockdiff: 0.000000 s (relative to rank 0)

MPI_Init:-:140733540331548:140733540331536:1305703565740309
MPI_Comm_rank:1305703565740336:6575584,0,4:140733541131592:1305703565740349
MPI_Comm_size:1305703565740361:6575584,0,4:140733541131588:1305703565740376
MPI_Send:1305703565740388:140733540331584:100000:1,4,4:1:0:6575584,0,4:1305703565742108
MPI_Recv:1305703565742124:140733540331584:100000:1,4,4:1:0:6575584,0,4:140733540331552:1305703565742420
MPI_Send:1305703565742430:140733540731584:100000:1,4,4:1:0:6575584,0,4:1305703565743819
MPI_Recv:1305703565743830:140733540731584:100000:1,4,4:1:0:6575584,0,4:140733540331552:1305703565744106
MPI_Barrier:1305703565744121:6575584,0,4:1305703565744164
MPI_Allgather:1305703565744178:140733540731584:300:1,4,4:140733540331584:300:1,4,4:6575584,0,4:1305703565747705
MPI_Barrier:1305703565747715:6575584,0,4:1305703565747750
MPI_Allreduce:1305703565747763:140733540731584:140733540331584:400:1,4,4:3:6575584,0,4:1305703565747828
MPI_Barrier:1305703565747838:6575584,0,4:1305703565747865
MPI_Allgather:1305703565747873:140733540731584:600:1,4,4:140733540331584:600:1,4,4:6575584,0,4:1305703565747926
MPI_Barrier:1305703565747935:6575584,0,4:1305703565747958
MPI_Allreduce:1305703565747966:140733540731584:140733540331584:800:1,4,4:3:6575584,0,4:1305703565748020
MPI_Barrier:1305703565748029:6575584,0,4:1305703565748053
MPI_Allgather:1305703565748061:140733540731584:900:1,4,4:140733540331584:900:1,4,4:6575584,0,4:1305703565748120
MPI_Barrier:1305703565748128:6575584,0,4:1305703565748147
MPI_Allreduce:1305703565748155:140733540731584:140733540331584:1200:1,4,4:3:6575584,0,4:1305703565748222
MPI_Barrier:1305703565748231:6575584,0,4:1305703565748247
MPI_Allgather:1305703565748256:140733540731584:1200:1,4,4:140733540331584:1200:1,4,4:6575584,0,4:1305703565748331
MPI_Barrier:1305703565748339:6575584,0,4:1305703565748354
MPI_Allreduce:1305703565748362:140733540731584:140733540331584:1600:1,4,4:3:6575584,0,4:1305703565748434
MPI_Barrier:1305703565748443:6575584,0,4:1305703565748891
MPI_Allgather:1305703565748900:140733540731584:1500:1,4,4:140733540331584:1500:1,4,4:6575584,0,4:1305703565748971
MPI_Barrier:1305703565748980:6575584,0,4:1305703565749002
MPI_Allreduce:1305703565749010:140733540731584:140733540331584:2000:1,4,4:3:6575584,0,4:1305703565749110
MPI_Barrier:1305703565749119:6575584,0,4:1305703565749141
MPI_Allgather:1305703565749149:140733540731584:1800:1,4,4:140733540331584:1800:1,4,4:6575584,0,4:1305703565749220
MPI_Barrier:1305703565749229:6575584,0,4:1305703565749262
MPI_Allreduce:1305703565749271:140733540731584:140733540331584:2400:1,4,4:3:6575584,0,4:1305703565749379
MPI_Barrier:1305703565749388:6575584,0,4:1305703565749403
MPI_Allgather:1305703565749412:140733540731584:2100:1,4,4:140733540331584:2100:1,4,4:6575584,0,4:1305703565749493
MPI_Barrier:1305703565749502:6575584,0,4:1305703565749531
MPI_Allreduce:1305703565749539:140733540731584:140733540331584:2800:1,4,4:3:6575584,0,4:1305703565763274
```

MPI_Barrier:1305703565763284:6575584,0,4:1305703565763389
MPI_Allgather:1305703565763398:140733540731584:2400:1,4,4:140733540331584:2400:1,4,4:6575584,0,4:1305703565763483
MPI_Barrier:1305703565763492:6575584,0,4:1305703565763523
MPI_Allreduce:1305703565763532:140733540731584:140733540331584:3200:1,4,4:3:6575584,0,4:1305703565763652
MPI_Barrier:1305703565763662:6575584,0,4:1305703565763693
MPI_Allgather:1305703565763702:140733540731584:2700:1,4,4:140733540331584:2700:1,4,4:6575584,0,4:1305703565763787
MPI_Barrier:1305703565763797:6575584,0,4:1305703565763826
MPI_Allreduce:1305703565763834:140733540731584:140733540331584:3600:1,4,4:3:6575584,0,4:1305703565764356
MPI_Barrier:1305703565764372:6575584,0,4:1305703565764387
MPI_Allgather:1305703565764396:140733540731584:3000:1,4,4:140733540331584:3000:1,4,4:6575584,0,4:1305703565764494
MPI_Barrier:1305703565764503:6575584,0,4:1305703565764525
MPI_Allreduce:1305703565764533:140733540731584:140733540331584:4000:1,4,4:3:6575584,0,4:1305703565764679
MPI_Barrier:1305703565764689:6575584,0,4:1305703565764704
Finalize clockdiff: 0.000000
MPI_Finalize:1305703565772457:-

Appendix I

Source code of another little MPI application:

```
#include <mpi.h>
#include <stdio.h>

#define DATASIZE 100000

int main(int argc, char **argv) {
  int rank;
  int procs; //number of mpi procs
  int i[DATASIZE]; //just some buffer
  int f[DATASIZE];
  int ret; //just a counter
  MPI_Status status;
  MPI_Request request, request2;
  i[0] = 6;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &procs);

  int left = rank - 1;
  if (left < 0) left = procs - 1;

  MPI_Irecv(i, 10, MPI_INT, left, 123, MPI_COMM_WORLD, &request);
  MPI_Send(f, 10, MPI_INT, (rank + 1) % procs, 123,
MPI_COMM_WORLD);
  MPI_Wait(&request, &status);

  MPI_Allreduce(i, f, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
```

```
  printf("%d done %d\n", rank, f[0]);
  fflush(stdout);

  MPI_Finalize();

  return 0;
}
```

The two traces collected when running the application above:

```
# htor's PMPI Tracer 0.9 Output File
# time: Thu May 26 09:31:39 2011
# hostname: compute-0-13.local.(none)
# uname: Linux compute-0-13.local 2.6.18-164.6.1.el5 #1 SMP Tue Nov 3 16:12:36 EST 2009 x86_64
# clockdiff: 0.000000 s (relative to rank 0)

MPI_Init:-:140734244848220:140734244848208:1306395099522160
MPI_Comm_rank:1306395099522192:6575584,0,2:140734245648280:1306395099522204
MPI_Comm_size:1306395099522216:6575584,0,2:140734245648276:1306395099522226
MPI_Irecv:1306395099522239:140734245248272:10:1,4,4:1:123:6575584,0,2:140734244848232:1306395099522273
MPI_Send:1306395099522286:140734244848272:10:1,4,4:1:123:6575584,0,2:1306395099522305
MPI_Wait:1306395099522318:140734244848232:140734244848240:1306395099522335
MPI_Allreduce:1306395099522348:140734245248272:140734244848272:1:1,4,4:3:6575584,0,2:1306395099522391
# Finalize clockdiff: 0.000000
MPI_Finalize:1306395099528881:-


# htor's PMPI Tracer 0.9 Output File
# time: Thu May 26 09:31:39 2011
# hostname: compute-0-6.local.(none)
# uname: Linux compute-0-6.local 2.6.18-164.6.1.el5 #1 SMP Tue Nov 3 16:12:36 EST 2009 x86_64
# clockdiff: 801.086426 s (relative to rank 0)

MPI_Init:-:140734388437676:140734388437664:1306395099521319
MPI_Comm_rank:1306395099521347:6575584,1,2:140734389237736:1306395099521360
MPI_Comm_size:1306395099521372:6575584,1,2:140734389237732:1306395099521381
MPI_Irecv:1306395099521393:140734388837728:10:1,4,4:0:123:6575584,1,2:140734388437688:1306395099521428
MPI_Send:1306395099521441:140734388437728:10:1,4,4:0:123:6575584,1,2:1306395099521460
MPI_Wait:1306395099521473:140734388437688:140734388437696:1306395099521505
MPI_Allreduce:1306395099521517:140734388837728:140734388437728:1:1,4,4:3:6575584,1,2:1306395099521595
# Finalize clockdiff: 798.940659
MPI_Finalize:1306395099528067:-
```

The .goal schedule produced out of the traces above using delays equal to function durations:

```
num_ranks 2

rank 0 {
l1: recv 40b from 1 tag 123
l2: calc 79000000
l1 requires l2
l3: calc 33967818
l1 requires l3
l3 requires l2
l4: send 40b to 1 tag 123
l5: calc 13000000
l4 requires l5
l5 irequires l1
l6: calc 18970000
l4 requires l6
l6 requires l5
l7: calc 17000000
l7 requires l1
```

```
l8: calc 13000000
l7 requires l8
l8 requires l4
l9: send 4b to 1 tag 1000000
l10: recv 4b from 1 tag 1000000
l11: calc 42975291
l11 requires l9
l11 requires l10
l12: calc 13000000
l9 requires l12
l10 requires l12
l12 requires l7
l13: calc 6490000000
l13 requires l11
}

rank 1 {
l1: recv 40b from 0 tag 123
l2: calc 74000000
l1 requires l2
l3: calc 34967818
l1 requires l3
l3 requires l2
l4: send 40b to 0 tag 123
l5: calc 13000000
l4 requires l5
l5 irequires l1
l6: calc 18970000
l4 requires l6
l6 requires l5
l7: calc 32000000
l7 requires l1
l8: calc 13000000
l7 requires l8
l8 requires l4
l9: send 4b to 0 tag 1000000
l10: recv 4b from 0 tag 1000000
l11: calc 77975291
l11 requires l9
l11 requires l10
l12: calc 12000000
l9 requires l12
l10 requires l12
l12 requires l7
l13: calc 6472000000
l13 requires l11
}
```

Appendix J

An example of the complete LogGOPSim output running a simulation which is not supposed to give the correct result:

```
vladimz@Computer3:~/omnetpp-4.1/samples/test$ ./LogGOPSim -f example_wrong.bin -L 0 -g 0 -G 0 -o 0 --scaleo
0 --ro 0 --rO 0 -v
key: 1107363766
```

```
omnet pid: 3272
size: 2 (1 CPUs, 1 NICs); L=0, o=0 g=0, G=0, eager=65535
init 0 (0,0) loclop: 79000000
init 1 (0,0) loclop: 74000000
[0] found loclop of length 79000000 - t: 0 (CPU: 0)
0 (0,0) loclop: 33966400, time: 79000000, offset: 2
[1] found loclop of length 74000000 - t: 0 (CPU: 0)
1 (0,0) loclop: 34966400, time: 74000000, offset: 2
[1] found loclop of length 34966400 - t: 74000000 (CPU: 0)
1 (0,0) recvs from: 0, tag: 123, size: 40, time: 108966400, offset: 0
[0] found loclop of length 33966400 - t: 79000000 (CPU: 0)
0 (0,0) recvs from: 1, tag: 123, size: 40, time: 112966400, offset: 0
[1] found recv from 0 - t: 108966400 (CPU: 0)
-- satisfy local irequires
++ [1] searching matching queue for src 0 tag 123
-- not found in local UQ -- add to RQ
1 (0,0) loclop: 13000000, time: 108966400, offset: 4
[1] found loclop of length 13000000 - t: 108966400 (CPU: 0)
1 (0,0) loclop: 18968000, time: 121966400, offset: 5
[0] found recv from 1 - t: 112966400 (CPU: 0)
-- satisfy local irequires
++ [0] searching matching queue for src 1 tag 123
-- not found in local UQ -- add to RQ
0 (0,0) loclop: 13000000, time: 112966400, offset: 4
[0] found loclop of length 13000000 - t: 112966400 (CPU: 0)
0 (0,0) loclop: 18968000, time: 125966400, offset: 5
[1] found loclop of length 18968000 - t: 121966400 (CPU: 0)
1 (0,0) send to: 0, tag: 123, size: 40, time: 140934400, offset: 3
[0] found loclop of length 18968000 - t: 125966400 (CPU: 0)
0 (0,0) send to: 1, tag: 123, size: 40, time: 144934400, offset: 3
[1] found send to 0 - t: 140934400 (CPU: 0)
-- satisfy local irequires
-- [1] send inserting msg to 0, t: 18446744073709551615
-- [1] eager -- satisfy local requires at t: 140934400
1 (0,0) loclop: 13000000, time: 140934400, offset: 7
[1] found loclop of length 13000000 - t: 140934400 (CPU: 0)
[0] found send to 1 - t: 144934400 (CPU: 0)
-- satisfy local irequires
-- [0] send inserting msg to 1, t: 18446744073709551615
-- [0] eager -- satisfy local requires at t: 144934400
0 (0,0) loclop: 13000000, time: 144934400, offset: 7
[0] found loclop of length 13000000 - t: 144934400 (CPU: 0)
[0] found msg from 1, t: 18446744073709551615 (CPU: 0)
.-- msg o,g not available -- reinserting
[0] found msg from 1, t: 157934400 (CPU: 0)
-- msg o,g available (nexto: 157934400, nextgr: 0)
++ [0] searching matching queue for src 1 tag 123
-- found in RQ
0 (0,0) loclop: 17000000, time: 157934400, offset: 6
[0] found loclop of length 17000000 - t: 157934400 (CPU: 0)
0 (0,0) loclop: 13000000, time: 174934400, offset: 11
[0] found loclop of length 13000000 - t: 174934400 (CPU: 0)
0 (0,0) send to: 1, tag: 1000000, size: 4, time: 187934400, offset: 8
0 (0,0) recvs from: 1, tag: 1000000, size: 4, time: 187934400, offset: 9
[0] found send to 1 - t: 187934400 (CPU: 0)
-- satisfy local irequires
```

```
-- [0] send inserting msg to 1, t: 18446744073709551615
-- [0] eager -- satisfy local requires at t: 187934400
[0] found recv from 1 - t: 187934400 (CPU: 0)
-- satisfy local irequires
++ [0] searching matching queue for src 1 tag 1000000
-- not found in local UQ -- add to RQ
[1] found msg from 0, t: 18446744073709551615 (CPU: 0)
.-- msg o,g not available -- reinserting
[1] found msg from 0, t: 153934400 (CPU: 0)
-- msg o,g available (nexto: 153934400, nextgr: 0)
++ [1] searching matching queue for src 0 tag 123
-- found in RQ
1 (0,0) loclop: 32000000, time: 153934400, offset: 6
[1] found loclop of length 32000000 - t: 153934400 (CPU: 0)
1 (0,0) loclop: 12000000, time: 185934400, offset: 11
[1] found loclop of length 12000000 - t: 185934400 (CPU: 0)
1 (0,0) send to: 0, tag: 1000000, size: 4, time: 197934400, offset: 8
1 (0,0) recvs from: 0, tag: 1000000, size: 4, time: 197934400, offset: 9
[1] found send to 0 - t: 197934400 (CPU: 0)
-- satisfy local irequires
-- [1] send inserting msg to 0, t: 18446744073709551615
-- [1] eager -- satisfy local requires at t: 197934400
[1] found recv from 0 - t: 197934400 (CPU: 0)
-- satisfy local irequires
++ [1] searching matching queue for src 0 tag 1000000
-- not found in local UQ -- add to RQ
[1] found msg from 0, t: 18446744073709551615 (CPU: 0)
.-- msg o,g not available -- reinserting
[1] found msg from 0, t: 197934400 (CPU: 0)
-- msg o,g available (nexto: 197934400, nextgr: 153934400)
++ [1] searching matching queue for src 0 tag 1000000
-- found in RQ
1 (0,0) loclop: 77934400, time: 197934400, offset: 10
[1] found loclop of length 77934400 - t: 197934400 (CPU: 0)
1 (0,0) loclop: 6472000000, time: 275868800, offset: 12
[1] found loclop of length 6472000000 - t: 275868800 (CPU: 0)
[0] found msg from 1, t: 18446744073709551615 (CPU: 0)
.-- msg o,g available (nexto: 187934400, nextgr: 157934400)
++ [0] searching matching queue for src 1 tag 1000000
-- found in RQ
0 (0,0) loclop: 42934400, time: 198048000, offset: 10
[0] found loclop of length 42934400 - t: 198048000 (CPU: 0)
0 (0,0) loclop: 6490000000, time: 240982400, offset: 12
[0] found loclop of length 6490000000 - t: 240982400 (CPU: 0)
PERFORMANCE: Processes: 2      Events: 30          Time: 4 s          Speed: 7.50 ev/s
Times:
Host 0: 6730982400
Host 1: 6747868800
```